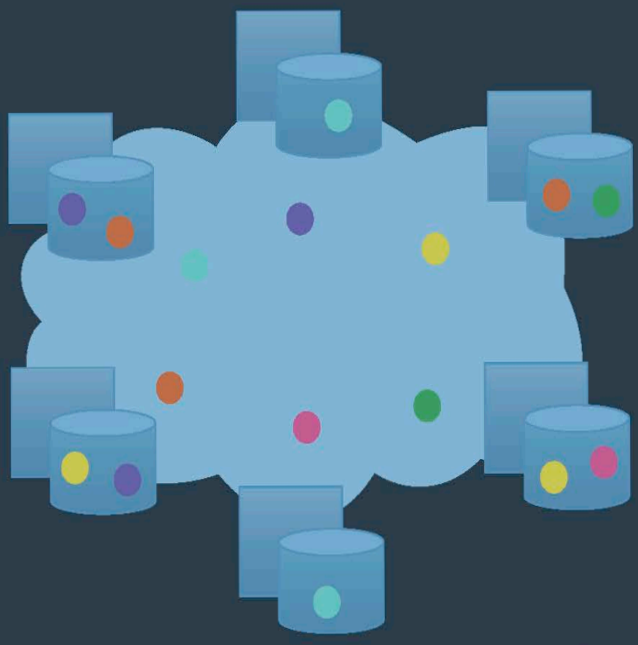
М. Тамер Ёсу, Патрик Вальдуриес

Принципы организации распределенных баз данных



Springer

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Принципы организации распределенных баз данных

М. Тамер Ёсу, Патрик Вальдуриес

Принципы организации распределенных баз данных

Principles of Distributed Database Systems

Fourth Edition

M. Tamer Özsu, Patrick Valduriez

Springer

Принципы организации распределенных баз данных

М. Тамер Ёсу, Патрик Вальдуриес

ДМК Пресс
издательство
Москва, 2021

УДК 004.655
ББК 32.973.26-018.2
Е81

Ёсу М. Т., Вальдуриес П.
Е81 Принципы организации распределенных баз данных / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2021. – 672 с.: ил.

ISBN 978-5-97060-391-8

В книге представлено подробное описание распределенных и параллельных баз данных с учетом новейших технологий. Авторы затрагивают такие темы, как проектирование распределенных и параллельных БД, контроль распределенных данных, распределенная обработка запросов и транзакций, интеграция баз данных. Отдельная глава посвящена обработке больших данных (в частности, обсуждаются распределенные системы хранения, потоковая обработка данных, платформы MapReduce и Spark, анализ графов и озера данных). Обработка веб-данных рассматривается с акцентом на технологию RDF, получившую широкое распространение.

В конце глав 2–12 приводятся упражнения, позволяющие закрепить теоретический материал. На сопроводительном сайте читатели найдут информацию об основах реляционных баз данных, обработке запросов, управлении транзакциями и компьютерных сетях. Кроме того, на сайте выложены все рисунки к книге, слайды и решения упражнений (только для преподавателей).

Издание может использоваться в качестве учебника для студентов и магистрантов, изучающих информатику и смежные дисциплины, а также заинтересует всех, кто занимается компьютерными науками.

УДК 004.655
ББК 32.973.26-018.2

ISBN 978-3-030-26252-5 (англ.)
ISBN 978-5-97060-391-8 (рус.)

Посвящается нашим семьям и родителям

Содержание

Об авторах

М. Мамер Ёсу – профессор Черитонской школы компьютерных наук в университете Ватерлоо в Канаде. Исследованиями в области распределенного управления данными он занимается уже тридцать лет. Состоит членом Королевского общества Канады, Американской ассоциации содействия развитию науки (AAAS), Ассоциации по вычислительной технике (ACM) и Института инженеров по электротехнике и электронике (IEEE). Является избранным членом Турецкой академии наук и членом общества «Сигма Кси». Является лауреатом премии за прижизненные достижения Канадского общества компьютерных наук за 2019 год, премии за проверенные временем достижения ACM SIGMOD за 2015 год, премии за вклад в науку ACM SIGMOD за 2008 год и премии выдающимся выпускникам технического колледжа университета штата Огайо за 2008 год. Также получил две премии за лучшую работу и один похвальный отзыв на публикацию. Состоит в редколлегиях многих журналов и книжных серий, наряду с Линь Лю является одним из главных редакторов «Энциклопедии по системам баз данных».

Патрик Вальдуриес – главный научный сотрудник французской компании Inria. Преподавал информатику в университете Пьера и Мари Кюри (UPMC) в Париже (2000–2002) и занимал должность исследователя в компании Microelectronics and Computer Technology Corp. в Остине, штат Техас (1985–1989). Начиная с 2019 года является научным консультантом стартапа LeanXcale.

В настоящее время возглавляет команду Zenith (включающую сотрудников компании Inria и университета Монпелье, LIRMM), которая занимается наукой о данных, в частности управлением данными в крупномасштабных распределенных и параллельных системах и управлением научными данными. Является заместителем редактора в нескольких журналах, в частности *VLDB Journal*, *Distributed and Parallel Databases* и *Internet and Databases*.

Занимал место в правлении таких крупных конференций, как SIGMOD и VLDB. Исполнял обязанности председателя конференций SIGMOD 2004, EDBT 2008 и VLDB 2009. Получил несколько наград за лучшую работу, в т. ч. на VLDB 2000. Лауреат премии по информатике от французского подразделения IBM за 1993 год и премии компании Inria, Французской академии наук и компании Dassault Systems за инновации в 2014 году. Является действительным членом ACM.

Предисловие

Первое издание этой книги вышло в 1991 году, когда технология была новой, а продуктов не так много. В предисловии к первому изданию мы цитировали Майкла Стоунбрейкера, который в 1988 году говорил, что в следующие 10 лет централизованные СУБД станут «антикварной редкостью», а большая часть организаций перейдет на распределенные СУБД. Это предсказание, конечно же, сбылось, в современном мире значительная доля систем либо распределенные, либо параллельные – обычно для их описания употребляется термин «горизонтально масштабируемые». Когда мы собирали материал для первого издания, курсы по базам данных для студентов и магистрантов были не так распространены, как сейчас, поэтому книга содержала пространные сведения о централизованных решениях, предварявшие обсуждение распределенных и параллельных систем. Но и на этом фронте произошли большие перемены, теперь трудно встретить магистранта, не имеющего хотя бы начальных знаний о технологии баз данных. Поэтому учебник по технологии распределенных и параллельных баз данных для магистрантов сейчас нужно позиционировать иначе. Такую цель мы и поставили себе в этом издании, сохранив, впрочем, многие новые темы, появившиеся в третьем издании. Перечислим основные изменения, внесенные в четвертое издание.

1. С годами побудительные мотивы этой технологии и среда ее развертывания претерпели изменения (веб, облако и т. д.). Поэтому вводная глава нуждалась в серьезном пересмотре. Мы переписали введение, отразив современный взгляд на технологию.
2. Мы добавили новую главу, посвященную обработке больших данных, в которую включили распределенные системы хранения, потоковую обработку данных, платформы MapReduce и Spark, анализ графов и озера данных. По мере распространения таких систем приобретает важность систематическое изложение этих вопросов.
3. Аналогично мы учли растущее влияние NoSQL-систем, посвятив им отдельную главу. В ней дается обзор четырех типов баз данных NoSQL (хранилища ключей и значений, документные хранилища, столбцовые базы данных и графовые СУБД), а также NewSQL-систем и полихранилищ.
4. Мы объединили главы об интеграции баз данных и обработке запросов к нескольким базам в одну главу.
5. Мы кардинально переработали изложение вопроса об обработке данных в вебе, сместив акцент с XML на технологию RDF, которая сейчас больше распространена. В эту главу мы включили обсуждение подходов к интеграции веб-данных, в т. ч. важный вопрос о качестве данных.
6. Мы также подвергли ревизии главу об одноранговой обработке данных и включили подробное объяснение технологии блокчейн.
7. Стремясь вычистить предыдущие главы, мы ужали главы, относящиеся к обработке запросов и управлению транзакциями, исключив принципиально централизованные методы и сосредоточив внимание на рас-

пределенных и параллельных. Попутно мы включили некоторые темы, которые приобрели большое значение, в частности динамическую обработку запросов (вихревых операторов), а также алгоритм консенсуса Paxos и его применение в протоколах фиксации.

8. Мы исправили главу о параллельных СУБД, уточнив цели, в частности пояснили различие между горизонтальным и вертикальным масштабированием и обсудили параллельные архитектуры, в т. ч. UMA и NUMA. Также добавлен новый раздел о параллельных алгоритмах сортировки и вариантах параллельных алгоритмов соединения, в которых задействованы преимущества памяти большого объема и многоядерных процессоров, которые ныне распространены повсеместно.
9. Мы пересмотрели главу о проектировании распределенности, включив пространное обсуждение современных подходов, сочетающих фрагментацию и размещение. После реорганизации материала эта глава стала центральным источником информации по секционированию данных во всех обсуждениях распределенного и параллельного управления данными.
10. Хотя объектные технологии по-прежнему играют роль в информационных системах, их значимость в системах распределенного и параллельного управления данными снизилась. Поэтому в этом издании мы исключили главу об объектных базах данных.

Как и в случае предыдущих изданий, в редактировании книги нам помогали многие коллеги, которым мы выражаем благодарность (не придерживаясь при перечислении какого-то определенного порядка). Дэн Олтеану (Dan Olteanu) предложил в главе 3 изящное обсуждение двух оптимизаций, которые могут значительно уменьшить время обслуживания материализованных представлений. Фил Бернштейн (Phil Bernstein) предоставил предварительные материалы к новым статьям о многоверсионном управлении транзакциями, легшие в основу обновленного обсуждения этой темы в главе 5. Хузаима Дауджи (Khuzaima Daudjee) также помог в подготовке списка более современных публикаций по распределенной обработке транзакций, который мы включили в библиографические ссылки к той же главе. Рикардо Хименес-Перис (Ricardo Jimenez-Peris) предоставил материал по высокопроизводительным транзакционным системам, включенный туда же. Деннис Шаша (Dennis Shasha) отредактировал новый раздел по блокчейну в главе по одноранговым системам. Майкл Кэри (Michael Carey) прочитал главы по большим данным, NoSQL, NewSQL, полихранилищам и параллельным СУБД и дал весьма подробные замечания, благодаря которым эти главы стали значительно лучше. Студенты Тамера, Анли Пачачи (Anil Pacaci), Халед Аммар (Khaled Ammar) и аспирант Сяофей Чжан (Xiaofei Zhang) написали подробные рецензии на главу по большим данным, и части их текстов включены в эту главу. В главу по NoSQL, NewSQL и полихранилищам включены части публикаций Бояна Колева (Boyan Kolev) и студентки Патрика Карлины Бондиомбой (Carlyna Bondiombouy). Джим Веббер (Jim Webber) отредактировал раздел по Neo4j этой главы. Характеристика графовых аналитических систем, приведенная в этой главе, частично основана на магистерской диссертации Миньянь Хана, в которой он заодно предложил подход на основе GiraphUC, обсуждаемый там же. Части этой главы прочитали и оставили весьма полезные замечания

также Семих Салихоглу (Semih Salihoglu) и Лукаш Голаб (Lukasz Golab). Алон Халеви поделился комментариями по поводу обсуждения веб-таблиц в главе 12. Обсуждение качества данных в процессе интеграции веб-данных написали Ихаб Ильяс (Ihab Ilyas) и Су Чу (Xu Chu). Стратос Идреос (Stratos Idreos) пояснил, как можно использовать крекинг базы данных в качестве подхода к секционированию, его текст включен в главу 2. Ренан Соуза (Renan Souza) и Фабиан Штётер (Fabian Stöter) просмотрели всю книгу целиком.

В третье издание книги было включено много новых тем, которые перекочевали в это издание, при написании этих глав большой вклад внесли многие наши коллеги. Мы хотим еще раз отметить их содействие, поскольку его влияние прослеживается и в новом издании. Рене Миллер (Renée Miller), Эрхард Рам (Erhard Rahm) и Алон Халеви (Alon Halevy) многое сделали для сведения воедино материалов по интеграции баз данных, которые затем были внимательно отрецензированы Авигдором Галом (Avigdor Gal). Матиас Ярке (Matthias Jarke), Сянь Ли (Xiang Li), Готфрид Фоссен (Gottfried Vossen), Эрхард Рам и Андреас Тор (Andreas Thor) предложили упражнения к этой главе. Хуберт Нааке (Hubert Naacke) внес вклад в раздел о моделировании гетерогенной стоимости, а Фабио Порто (Fabio Porto) – в раздел об адаптивной обработке запросов. Главу 6 о репликации данных невозможно было бы написать без помощи Густаво Алонсо (Gustavo Alonso) и Беттины Кемме (Bettina Kemme). Эстер Пачитти (Esther Pacitti) также внесла вклад в главу о репликации данных – прочитав ее и предложив вводные материалы; она же помогала в написании раздела о кластерах баз данных в главе о параллельных СУБД. При работе над главой об одноранговой обработке данных мы много беседовали с Бень Чин Ои (Beng Chin Ooi). В разделе этой главы, посвященном обработке запросов в одноранговых системах, использованы материалы из докторской диссертации Резы Акбаринья (Reza Akbarinia) и Венцеслао Пальма (Wenceslao Palma), а в разделе о репликации – материалы из докторской диссертации Видала Мартинса (Vidal Martins).

Мы благодарны нашему редактору в издательстве Springer Сюзан Лагерстром-Файф (Susan Lagerstrom-Fife) за продвижение проекта в самом издательстве и за постоянные напоминания о необходимости закончить работу вовремя. Мы пропустили почти все назначенные ей крайние сроки, но надеемся, что результат получился неплохим.

Наконец, нам было бы очень интересно услышать ваши замечания и предложения. Мы приветствуем любые отзывы, но особенно нас интересует ваше мнение по следующим вопросам:

1) любые ошибки и опечатки, просочившиеся, несмотря на все наши усилия (хочется надеяться, что их немного);
2) темы, которые следовало бы исключить, и, наоборот, темы, которые нужно добавить или изложить более подробно;
3) придуманные вами упражнения, которые вы хотели бы включить в книгу.

Ватерлоо, Канада — М. Тамер Ёсу (tamer.ozsu@uwaterloo.ca)
Монпелье, Франция — Патрик Вальдуриес (patrick.valduriez@inria.fr)
Июнь 2019

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Springer очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Глава 1

Введение

Современная вычислительная среда в значительной степени распределенная – компьютеры подключены к интернету и образуют всемирную распределенную систему. В организациях имеются территориально распределенные связанные между собой центры обработки данных (ЦОД), насчитывающие сотни, а то и тысячи компьютеров, объединенных в высокоскоростную сеть, которая содержит как распределенные, так и параллельные системы (рис. 1.1). Объем данных, хранимых в такой среде, возрос многократно. Не все данные хранятся в системах баз данных (на самом деле там хранится лишь малая их часть), но возникает желание так или иначе управлять этим распределенным по большой территории массивом данных. Это и есть задача распределенных и параллельных систем баз данных, которые несколько десятилетий назад занимали лишь небольшую нишу в мировой вычислительной среде, а теперь вышли на авансцену. В данной главе мы дадим общий обзор этой технологии, а в последующих займемся деталями.

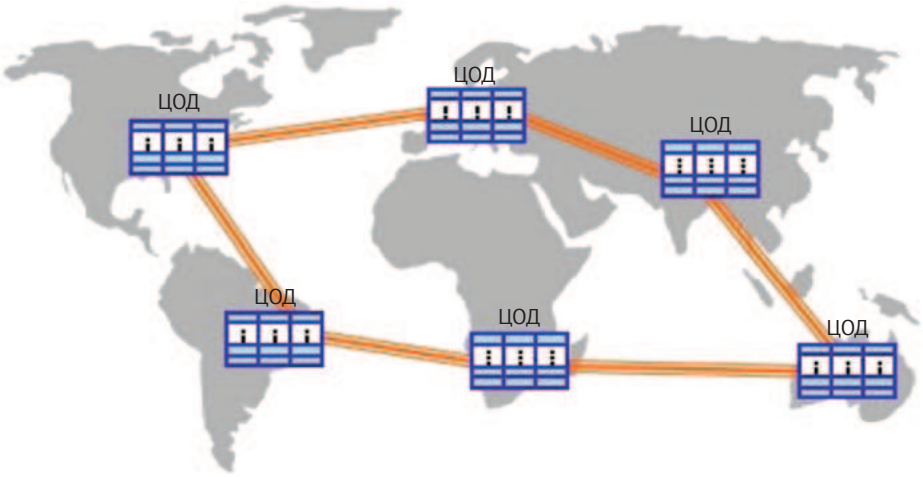


Рис. 1.1 ❖ Территориально распределенные центры обработки данных

1.1. Что такое распределенная система баз данных?

Распределенную базу данных мы определяем как *набор из нескольких логически взаимосвязанных баз данных, расположенных в узлах распределенной системы*. *Распределенной системой управления базами данных* (распределенной СУБД) мы далее называем *программную систему, которая допускает управление распределенной базой данных и делает распределенность прозрачной для пользователей*. Иногда, говоря «распределенная система баз данных» (распределенная СУБД), имеют в виду как распределенную базу данных, так и распределенную СУБД в нашем понимании. Мы выделяем две важные характеристики: логическую взаимосвязанность данных и их нахождение в распределенной системе.

Существование распределенной системы – важный момент. В этом контексте под *распределенной вычислительной системой* понимается несколько связанных между собой автономных обрабатывающих элементов (ОЭ). Возможности обрабатывающих элементов могут различаться, они могут быть гетерогенными, связи между ними тоже могут быть различными, но важно то, что ОЭ не имеют прямого доступа к состоянию друг друга, а могут узнать его, лишь обмениваясь сообщениями и, следовательно, неся затраты на коммуникацию. Поэтому если данные распределены, то доступ к ним и управление ими логически непротиворечивым способом требуют особого внимания со стороны распределенной СУБД.

Распределенная СУБД – это не просто «набор файлов», которые можно по отдельности хранить в каждом ОЭ распределенной системы (обычно называемом «узлом» распределенной СУБД); данные в распределенной СУБД взаимосвязаны. Мы не будем конкретизировать, что означает «взаимосвязаны», поскольку требования зависят от типа данных. Например, в случае реляционных данных различные отношения или их части могут храниться в разных узлах (подробнее об этом см. главу 2), и для ответа на запросы, выражаемые, как правило, на языке SQL, требуется выполнять операции соединения или объединения. Обычно можно определить *схему* таких распределенных данных. На другом полюсе находятся данные в системах NoSQL (см. главу 11), в которых возможно гораздо менее ограничительное определение взаимосвязанности; например, это могут быть вершины графа, хранящиеся в разных узлах.

Подводя итоги этому обсуждению, можно сказать, что распределенная СУБД *логически едина*, но *физически распределена*. Это означает, что пользователь, работающий с распределенной СУБД, воспринимает ее как единую базу данных, хотя составляющие ее данные физически находятся в разных местах.

Как уже было сказано, обычно рассматриваются два типа распределенных СУБД: территориально распределенные (или *георапределенные*) и сосредоточенные в одном месте (одноузловые). В первом случае узлы соединены между собой глобальной сетью, для которой характерны большие задержки при передаче сообщений и более высокая частота ошибок. А во втором речь идет о системах, в которых ОЭ находятся близко друг к другу, так что

обмен сообщениями происходит гораздо быстрее (современные технологии позволяют считать задержку пренебрежимо малой) и с очень низкой частотой ошибок. Одноузловые распределенные СУБД обычно размещаются на кластерах компьютеров в одном ЦОДе и называются параллельными СУБД (их ОЭ по-английски называются «node» – в отличие от «site», а по-русски в обоих случаях употребляется термин «узел»). Выше отмечалось, что сегодня легко встретить распределенные СУБД, состоящие из нескольких одноузловых кластеров, соединенных глобальной сетью, т. е. имеет место гибридная многоузловая система. В этой книге мы в основном будем рассматривать задачи управления данными в узлах геораспределенной СУБД, а о проблемах одноузловых систем речь пойдет только в главах 8, 10 и 11, где обсуждаются параллельные СУБД, большие данные и системы NoSQL/NewSQL.

1.2. История распределенных СУБД

До появления систем баз данных в 1960-х годах каждое приложение само определяло свои данные и управляло ими (рис. 1.2). В этом режиме приложение принимало решения о структуре и методах доступа к данным и отвечало за управление файлом в системе хранения. Это приводило к значительной и неконтролируемой избыточности данных и высоким трудозатратам программистов, вынужденных заниматься управлением данными в приложениях.

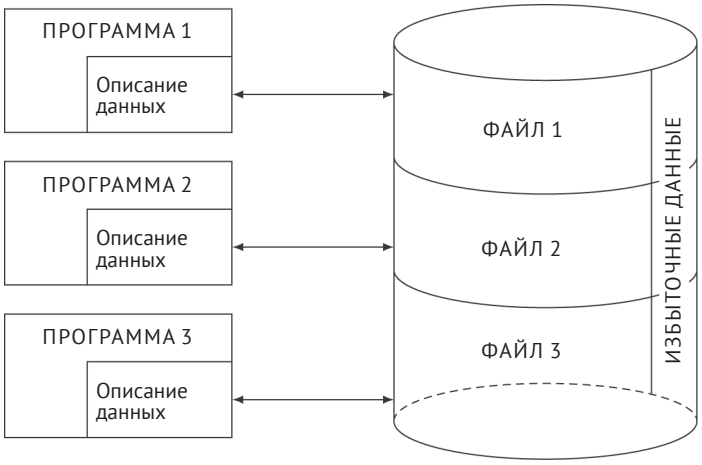


Рис. 1.2 ❖ Традиционная обработка файлов

Система баз данных позволяет определять и администрировать данные централизованно (рис. 1.3). Этот новый подход ведет к *независимости данных*, когда прикладная программа безразлична к изменению логической или физической организации данных, и наоборот. Таким образом, программисты освобождаются от ответственности за управление и сопровождение нужных им данных, а избыточность можно устранить (или уменьшить).

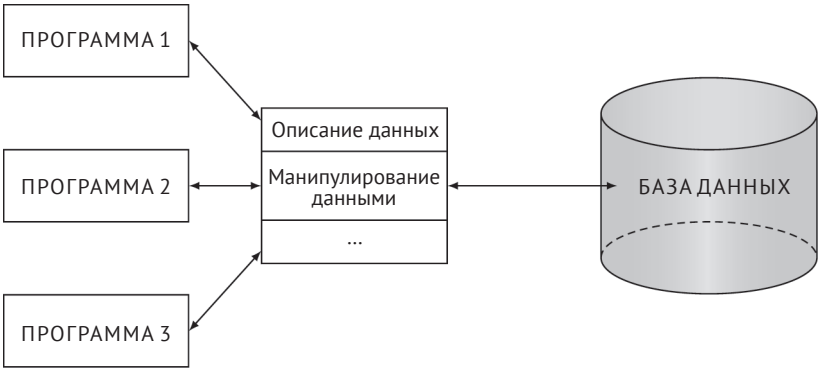


Рис. 1.3 ❖ Обработка базы даннх

Одним из побудительных мотивов использования систем баз данных являлось желание объединить всю информацию о работе предприятия и предоставить интегрированный и контролируемый доступ к этим данным. Мы сознательно употребляем термин «интегрированный», а не «централизованный», потому что данные, как уже было сказано, могут размещаться на разных территориально разнесенных компьютерах. Именно в этом смысл технологии распределенных баз данных. Мы уже отмечали, что физически распределенная система может находиться в одном месте или в нескольких местах. Поэтому каждый узел на рис. 1.5 может представлять собой центр обработки данных, соединенных сетью с другими центрами. Именно распределенные среды такого типа являются типичными в настоящее время, и именно их мы будем изучать в этой книге.

С годами архитектура распределенной системы баз данных претерпела значительные изменения. Такие ранние распределенные системы баз данных, как Distributed INGRES и SDD-1, проектировались как территориально распределенные системы с очень медленными сетевыми соединениями, поэтому они всеми силами стремились оптимизировать операции, чтобы уменьшить обмен данными по сети. Это были первые *одноранговые системы* (peer-to-peer – P2P) в том смысле, что все узлы имели похожую функциональность в части управления данными. С развитием персональных компьютеров и рабочих станций стала преобладать модель распределения типа *клиент-сервер*, в которой данные переместились на тыловой сервер, а пользовательские приложения работали на фронтальных рабочих станциях. Особенно часто такие системы стали развертываться на одной территории, где можно было обеспечить более высокую скорость сети и, стало быть, более частое взаимодействие между клиентами и сервером (или серверами). В 2000-х годах произошло возрождение P2P-систем, в которых исчезло различие между клиентскими и серверными компьютерами. Современные P2P-системы отличались от ранних в нескольких важных отношениях, которые мы обсудим ниже в этой главе. Все эти архитектуры существуют и сегодня и обсуждаются в последующих главах.

Становление Всемирной паутины (или просто веба) как основной платформы для совместной работы и обмена файлами оказало громадное влияние

на исследования по управлению распределенными данными. Стало доступно гораздо больше данных, но это не были тщательно структурированные и точно определенные данные, как в типичной СУБД; они были слабо структурированы или не структурированы вовсе (т. е. какую-то структуру они имели, но определенную не на уровне схемы базы данных), их происхождение было неизвестно (т. е. данные могли быть «грязными» или ненадежными), и зачастую они были противоречивы. Ко всему прочему многие данные хранились в системах, к которым не было простого доступа (в *скрытом вебе*). Поэтому усилия по распределенному управлению данными направляются на доступ к этим данным осмысленными способами.

Это направление развития стимулировало исследования в области *интеграции баз данных* – дисциплине, которая существовала с самого начала работ по распределенным базам данных. Первоначально эти усилия были направлены на поиск способов доступа к данным в различных базах (отсюда и термины *федеративная база данных* и *мультибаза данных*), но с появлением веб-данных фокус сместился в сторону виртуальной интеграции данных разных типов (поэтому термин *интеграция данных* стал более популярным). Сейчас в моде термин *озеро данных*, который подразумевает, что все данные собираются в логически едином хранилище, из которого каждое приложение извлекает нужные ему данные. Мы обсудим интеграцию данных в главе 7, а озера данных – в главах 10 и 12.

За последние десять лет важным явлением стали облачные вычисления. Под этим понимается вычислительная модель, в которой ряд поставщиков услуг предоставляют в общее пользование разделяемые и территориально распределенные вычислительные ресурсы, так что пользователи могут арендовать их по мере необходимости. Клиенты могут взять в аренду базовую вычислительную инфраструктуру для разработки собственного программного обеспечения, а затем решить, какую операционную систему предпочитают, и создать для себя виртуальные машины (ВМ) со средой, в которой хотят работать, – такой подход называется *инфраструктура как услуга* (IaaS). Возможна и более развитая облачная среда, в которой в аренду сдается не только базовая инфраструктура, но и вся вычислительная платформа, на которой клиенты разрабатывают свое ПО, – это *платформа как услуга* (PaaS). Самый продвинутый вариант – когда поставщик услуг предоставляет в аренду конкретное программное обеспечение, этот подход называется *программное обеспечение как услуга* (SaaS). В последнее время начинают предлагать услуги управления распределенной базой данных в облаке как часть SaaS.

Мы дадим общий обзор всех этих архитектур в разделе 1.6.1.2, а затем обсудим их в отдельных главах более подробно.

1.3. Различные способы доставки данных

В распределенных базах данных доставка данных производится между узлами – либо от серверных узлов клиентским в ответ на запросы, либо между несколькими серверными узлами. Мы будем характеризовать варианты

доставки данных по трем независимым осям: *режимы доставки*, *частота* и *методы коммуникации*. Комбинирование всех трех свойств дает проектировщику достаточно богатый выбор.

Существует три режима доставки: вытягивание, проталкивание и гибридный. В режиме вытягивания передача данных инициируется узлом в виде запроса поставщику данных – это может быть клиент, запрашивающий данные у сервера, или сервер, запрашивающий данные у другого сервера. Далее мы будем употреблять термины «получатель» и «поставщик» для обозначения компьютеров, которые получают и отправляют данные соответственно. Получив запрос, поставщик находит и передает данные. Основная характеристика доставки методом вытягивания заключается в том, что получатель узнаёт о наличии новых или обновленных данных у поставщика только после того, как явно спросит об этом. Кроме того, в режиме вытягивания работа поставщика постоянно прерывается новыми запросами. У этого режима есть ограничение – получатель имеет только те данные, о которых знает, и только тогда, когда попросит. Традиционные СУБД предлагают в основном доставку данных в режиме вытягивания.

В режиме проталкивания передачу данных инициирует поставщик без явного запроса. Основная трудность такого подхода – решить, какие данные будут представлять всеобщий интерес и когда отправлять их потенциально заинтересованным получателям: периодически, нерегулярно или по некоторому условию. Таким образом, полезность проталкивания сильно зависит от того, насколько точно поставщик умеет предсказывать нужды получателей. В режиме проталкивания поставщики распространяют информацию либо неограниченному кругу получателей (ненаправленное вещание), которые прослушивают передающую среду, либо избранному множеству получателей (групповое вещание), принадлежащему определенным категориям.

Гибридный режим сочетает механизмы вытягивания и проталкивания. Один из способов комбинирования того и другого – постоянный запрос (см. раздел 10.3), когда сначала в режиме вытягивания (посредством отправки запроса) инициируется первая передача данных от поставщика клиента, а затем поставщик уже сам отправляет обновленные данные в режиме проталкивания.

Существует три типичные частоты доставки данных, характеризующие ее регулярность: периодическая, условная и ситуативная (или нерегулярная).

При *периодической* доставке данные отправляются поставщиками с регулярными интервалами. Величина интервала может задаваться по умолчанию системой или в профиле каждого получателя. Периодическая доставка возможна как в режиме вытягивания, так и в режиме проталкивания. Она выполняется по заранее заданному регулярному расписанию. Примером периодического вытягивания является еженедельный запрос цены акций компании, а периодического проталкивания – отправка цен акций приложением по расписанию, скажем каждое утро. Периодическое проталкивание особенно полезно в ситуациях, когда получатель доступен не постоянно или не всегда может отреагировать на присланные данные, как, скажем, в случае, когда клиент установлен в мобильном устройстве, которое иногда отключается от сети.

В случае *условной доставки* данные отправляются поставщиком, когда выполнены некоторые условия, заданные в профиле получателей. Это может быть простое условие типа истечения промежутка времени или сложные правила вида событие–условие–действие. Условная доставка чаще всего используется в системах, работающих в гибридном режиме или в режиме вытягивания. В этом случае данные рассылаются при выполнении предопределенного условия, а не по расписанию. Примером условного проталкивания может служить приложение, которое отправляет цены акций, только когда они изменяются. Примером гибридной условной доставки является приложение, которое отправляет выписку об остатке на счете, только когда остаток оказывается на 5 % меньше оговоренного порога. Условное проталкивание предполагает, что изменения критически важны для получателей, которые постоянно ожидают их и должны отреагировать на полученные данные. Гибридное условное проталкивание дополнительно предполагает, что получатели могут смириться с пропуском какой-то информации об обновлениях.

Ситуативная доставка выполняется нерегулярно и в основном в системах, работающих в режиме вытягивания. Данные вытягиваются у поставщиков по ситуации в ответ на запросы. Напротив, периодическое вытягивание имеет место, когда запрашивающая сторона производит опрос поставщиков по расписанию.

Третья характеристика доставки информации – метод коммуникации. Он определяет, каким образом поставщики и получатели взаимодействуют с целью доставки информации клиентам. Вариантов два: одноадресная передача и передача один-ко-многим. В *одноадресном* режиме поставщик посылает данные одному получателю в определенном режиме доставки и с определенной частотой. При передаче один-ко-многим поставщик посылает данные сразу нескольким получателям. Отметим, что речь не идет о конкретном протоколе: в режиме один-ко-многим может применяться как протокол группового вещания, так и протокол широковещания.

Следует отметить, что описанная классификация – предмет споров. Неочевидно, что все точки пространства вариантов имеют смысл. Кроме того, придание смысла некоторым комбинациям, например условная **и** периодическая (быть может, и осмысленная), наталкивается на трудности. Однако в первом приближении она может охарактеризовать сложность систем управления распределенными данными. В этой книге нас будут интересовать в первую очередь системы ситуативной доставки данных в режиме вытягивания, но мы обсудим режим проталкивания и гибридный режим в разделе 10.3, посвященном потоковым системам.

1.4. Обещания распределенных СУБД

Можно привести много преимуществ распределенных СУБД, но в основном они сводятся к четырем фундаментальным положениям, которые можно рассматривать как обещания технологии: прозрачное управление распределенными и реплицированными данными, надежный доступ к данным по-

средством распределенных транзакций, улучшенная производительность и простое расширение системы. В этом разделе мы обсудим эти четыре обещания и попутно введем ряд понятий, который будем изучать в последующих главах.

1.4.1. Прозрачное управление распределенными и реплицированными данными

Под прозрачностью понимается отделение высокоуровневой семантики системы от низкоуровневых вопросов реализации. Иными словами, прозрачная система «скрывает» детали реализации от пользователей. Преимущество полностью прозрачной СУБД – высокий уровень поддержки, оказываемой разработке сложных приложений. Прозрачность в распределенных СУБД можно рассматривать как обобщение концепции независимости данных в централизованных СУБД (об этом еще будет сказано ниже).

Начнем обсуждение с примера. Рассмотрим конструкторскую компанию, имеющую отделения в Бостоне, Ватерлоо, Париже и Сан-Франциско. В каждом отделении выполняются какие-то проекты, и компания хочет иметь базу данных работников, проектов и связанных с этим данных. В предположении, что база данных реляционная, мы можем хранить эту информацию в нескольких отношениях (рис. 1.4): в EMP хранятся данные о работнике – номер, имя и должность[1], в PROJ – данные о проектах (атрибут LOC содержит отделение, в котором выполняется проект). В PAY хранятся сведения о зарплатах (в предположении, что два человека, занимающих одинаковые должности, получают одинаковую зарплату), а в ASG – сведения о распределении людей по проектам (DUR – продолжительность назначения, RESP – функция данного человека в данном проекте). Если бы все эти данные хранились в централизованной СУБД и мы хотели бы найти имена и зарплаты тех, кто работает над одним проектом больше 12 месяцев, то написали бы такой SQL-запрос:

```
SELECT ENAME, SAL
FROM EMP NATURAL JOIN ASG, EMP NATURAL JOIN PAY
WHERE ASG.DUR > 12
```

EMP(ENO, ENAME, TITLE)
PROJ(PNO, PNAME, BUDGET, LOC)
ASG(ENO, PNO, RESP, DUR)
PAY(TITLE, SAL)

Рис. 1.4 ❖ Пример базы данных конструкторской компании

Однако с учетом распределенной природы бизнеса этой компании предпочтительно локализовать данные, так чтобы сведения о работающих в офисе Ватерлоо хранились в Ватерлоо, сведения о работающих в бостонском офи-

[1] Первичные ключи подчеркнуты.

се – в Бостоне и т. д. То же самое относится к данным о проектах и зарплатах. Таким образом, дело свелось к процессу разбиения каждого отношения на части и хранению разных частей в разных узлах. Этот процесс называется *секционированием*, или *фрагментацией данных*, мы подробно рассмотрим его в главе 2.

Далее, иногда предпочтительно дублировать часть данных в других узлах ради повышения производительности и надежности. В результате мы получаем фрагментированную и реплицированную распределенную базу данных (рис. 1.5). Полностью прозрачный доступ означает, что пользователь может сформулировать запрос именно так, как показано выше, не обращая внимания на фрагментацию, местоположение или репликацию данных, и оставить решение этих вопросов системе. Чтобы система могла адекватно обработать запрос такого типа к распределенной, фрагментированной и реплицированной базе данных, она должна реализовывать различные типы прозрачности, о которых мы и поговорим ниже.

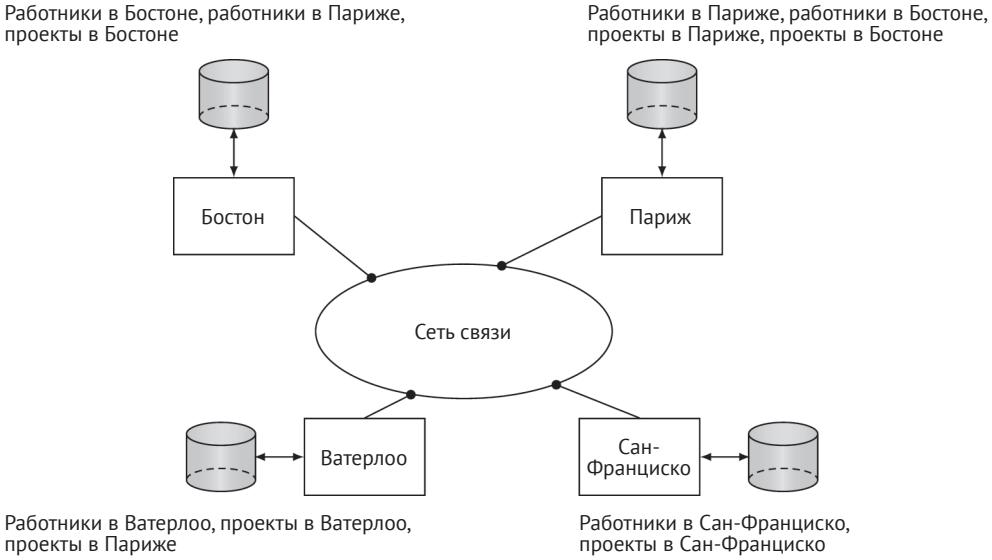


Рис. 1.5 ❖ Распределенная база данных

Независимость данных. Эта концепция, заимствованная у централизованных СУБД, относится к независимости пользовательских приложений от изменений в определении и организации данных, и наоборот.

Обычно говорят о двух типах независимости данных: логической и физической. Под *логической независимостью данных* понимается независимость приложений от изменений в логической структуре (т. е. схеме) базы данных. С другой стороны, *физическая независимость данных* связана с сокрытием деталей структуры хранения от пользовательских приложений. После того как приложение написано, его не должны волновать детали физической организации данных. Поэтому не должно возникать необходимости в модифи-

кации приложения, когда организация данных изменяется из соображений производительности.

Прозрачность сети. Желательно, чтобы пользователи были экранированы от деталей работы сети связи, соединяющей узлы, хорошо бы даже, чтобы само существование сети было скрыто. Тогда не будет разницы между приложениями, работающими с централизованной и распределенной базами данных. Такой вид прозрачности называется *прозрачностью сети*, или *прозрачностью распределения*.

Иногда выделяют два типа прозрачности распределения: прозрачность местоположения и прозрачность именования. Под *прозрачностью местоположения* понимается тот факт, что команда, необходимая для выполнения задачи, не зависит ни от местоположения данных, ни от системы, в которой операция выполняется. *Прозрачность именования* означает, что каждому объекту в базе данных присвоен уникальный идентификатор. В отсутствие прозрачности именования пользователи должны включать имя (или идентификатор) местоположения в состав имени объекта.

Прозрачность фрагментации. Как уже было сказано, часто желательно разбить каждое отношение базы данных на меньшие фрагменты и обращаться с каждым фрагментом как с отдельным объектом базы данных (т. е. отдельным отношением). Обычно это делают из соображений производительности, доступности и надежности – более подробное обсуждение см. в главе 2. Было бы хорошо, чтобы пользователи не знали о фрагментации при формулировании запросов, а система умела отображать запрос, сформулированный в терминах полных отношений, описанных в схеме, на множество запросов, выполняемых над частичными отношениями. Иными словами, требуется найти стратегию обработки запросов, основанную на фрагментах, а не на отношениях, пусть даже запросы формулируются в терминах последних.

Прозрачность репликации. Из соображений производительности, надежности и доступности обычно желательно иметь возможность распределять данные посредством репликации между машинами в сети. В предположении, что данные реплицируются, возникает вопрос, должны ли пользователи знать о существовании копий или система будет обрабатывать их самостоятельно, а пользователь должен действовать так, будто имеется всего одна копия данных (заметим, что мы говорим не о местоположении копий, а только об их существовании). С точки зрения пользователя, предпочтительно не думать о копиях и никак не упоминать тот факт, что некоторое действие может или должно применяться к нескольким копиям. Вопрос о репликации в распределенной базе данных поднимается в главе 2 и подробно обсуждается в главе 6.

1.4.2. Обеспечение надежности с помощью распределенных транзакций

Распределенные СУБД призваны повысить надежность, поскольку в них имеются реплицированные части, а значит, исключается единая точка отказа. Отказа одного узла или отказа линии связи, из-за которого один или не-

сколько узлов становятся недоступными, недостаточно для выхода из строя системы в целом. В случае распределенной базы данных это означает, что некоторые данные могут оказаться недоступными, но при должной аккуратности пользователям будет разрешено обращаться к другим частям распределенной базы данных. Под «должной аккуратностью» обычно понимается поддержка распределенных транзакций.

СУБД с полной поддержкой транзакций гарантирует, что одновременное выполнение транзакций несколькими пользователями не приведет к несогласованности базы данных, т. е. каждый пользователь может считать, что только его запрос выполняется базой (*прозрачность конкурентности*) даже при наличии отказов системы (*прозрачность отказов*), при условии что каждая транзакция корректна, т. е. не нарушает правил целостности, заданных в базе данных.

Для поддержки транзакций необходимо реализовать управление конкурентностью транзакций и распределенные протоколы обеспечения надежности, в частности протоколы двухфазной фиксации (2PC) и распределенного восстановления. Эти протоколы значительно сложнее, чем в централизованных СУБД, они обсуждаются в главе 5. Для поддержки реплик необходима реализация протоколов управления репликами, которые следят за соблюдением семантики доступа к ним. Эти протоколы обсуждаются в главе 6.

1.4.3. Повышенная производительность

Повышение производительности распределенной СУБД обычно обусловлено двумя моментами. Во-первых, распределенная СУБД фрагментирует базу данных, давая возможность хранить данные поблизости от точек их использования (*локальность данных*). Такой подход дает два преимущества:

1) поскольку каждый узел обрабатывает только часть базы данных, конкуренция за процессор и службы ввода-вывода не так сильна, как в централизованных базах;
2) благодаря локальности уменьшаются задержки удаленного доступа, обычно свойственные глобальным сетям.

Этот момент связан с накладными расходами на распределенные вычисления, когда данные находятся в удаленных узлах, к которым приходится обращаться по сети. Считается, что при таких условиях лучше приблизить функциональность управления данными к месту их расположения, чем передавать большие объемы данных. Иногда это положение вызывает споры. Некоторые возражают, что вследствие широкого распространения высокоскоростных сетей с высокой пропускной способностью распределение данных и функций управления ими потеряло смысл и проще хранить данные в центральном узле на очень большом компьютере и обращаться к ним по высокоскоростным сетям. Такой подход называется архитектурой с *вертикальным* масштабированием. Аргумент притягательный, но в нем упущена важная особенность распределенных баз данных. Во-первых, в большинстве современных приложений данные по факту распределенные; спорить можно только о том, как и где их обрабатывать. Во-вторых, и это даже важнее, в этой

аргументации не различается полоса пропускания сети (пропускная способность каналов связи) и задержка (сколько времени требуется для передачи данных). Задержка внутренне присуща распределенным средам, существуют физические ограничения на скорость передачи данных в компьютерных сетях. Удаленный доступ к данным может привести к задержкам, неприемлемым для многих приложений.

Второй момент – это то, что присущий распределенным системам параллелизм можно использовать для организации внутризапросного и межзапросного параллелизма. Под *межзапросным параллелизмом* понимается параллельное выполнение нескольких запросов, сгенерированных конкурентными транзакциями, с целью повысить транзакционную пропускную способность. Определение внутризапросного параллелизма различно в распределенных и параллельных СУБД. В первом случае внутризапросный параллелизм достигается путем разбиения одного запроса на несколько подзапросов, каждый из которых выполняется в отдельном узле и обращается к разным частям распределенной базы данных. В параллельных же СУБД это достигается за счет *межоператорного* и *внутриоператорного* параллелизма. Для реализации межоператорного параллелизма различные операторы в дереве запроса параллельно выполняются на разных процессорах, тогда как в случае внутриоператорного параллелизма один и тот же оператор выполняется несколькими процессорами, каждый из которых работает со своим подмножеством данных. Заметим, что обе эти формы параллелизма встречаются также в распределенной обработке запросов.

Внутриоператорный параллелизм основан на разложении одного оператора на множество независимых подоператоров, называемых *экземплярами оператора*. Это разложение производится с использованием секционирования отношений. Затем каждый экземпляр оператора обрабатывает одну секцию отношения. При разложении оператора часто удается воспользоваться начальным секционированием данных (например, если данные секционированы по атрибуту соединения). Для иллюстрации внутриоператорного параллелизма рассмотрим простой запрос SELECT с соединением. Оператор SELECT можно непосредственно разложить на несколько операторов SELECT, по одному для каждой секции, так что никакого перераспределения не требуется (рис. 1.6). Заметим, что если отношение секционировано по атрибуту выборки, то свойства секционирования можно использовать для исключения некоторых экземпляров SELECT. Например, в случае выборки по точному совпадению будет выполняться только один экземпляр SELECT, если отношение было секционировано по хешу (или по диапазону) атрибута выборки. Разложить оператор соединения JOIN сложнее. Чтобы соединения были независимы, каждую секцию одного отношения `R` можно соединить с другим отношением `S` целиком. Такое соединение будет крайне неэффективным (если только `S` не очень мало), потому что придется пересылать `S` на каждый участвующий в вычислении процессор. Более эффективно использовать свойства секционирования. Например, если `R` и `S` секционированы по хешу атрибута соединения и если имеет место операция эквисоединения, то мы можем секционировать соединение на независимые соединения. Это идеальный случай, но он не всегда применим, потому что зависит от началь-

ного секционирования R и S. В других случаях один или два операнда, возможно, придется пересекционировать. Наконец, можно заметить, что функция секционирования (хеширование, по диапазону, циклически – все они обсуждаются в разделе 2.3.1) не зависит от локального алгоритма (например, вложенные циклы, хеширование, сортировка слиянием), используемого для обработки оператора соединения на каждом процессоре. Например, для соединения хешированием с использованием секционирования хешированием нужно две хеш-функции. Первая, h_1, применяется для секционирования двух базовых отношений по атрибуту соединения. Вторая же, h_2, может быть разной на каждом процессоре и применяется для обработки соединения на этом процессоре.

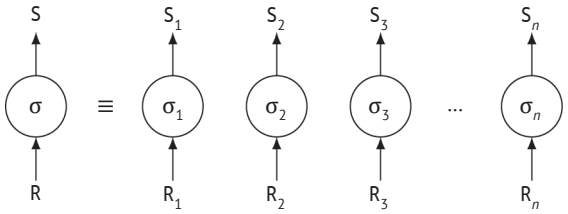


Рис. 1.6 ❖ Внутриоператорный параллелизм,
σ_i – *i*-й экземпляр оператора, *n* – степень параллелизма

Встречаются две формы межоператорного параллелизма. В случае *конвейерного параллелизма* несколько операторов, между которыми имеется связь вида производитель–потребитель, выполняются параллельно. Например, оба оператора выборки на рис. 1.7 будут выполняться параллельно с оператором соединения. Преимущество такого выполнения в том, что промежуточный результат необязательно материализовать целиком, поэтому экономится память и уменьшается число обращений к диску. *Независимый параллелизм* имеет место, когда между исполняемыми параллельно операторами нет никакой зависимости. Например, оба оператора выборки на рис. 1.7 можно выполнять параллельно. Такая форма параллелизма очень привлекательна, потому что между процессорами не возникает интерференции.

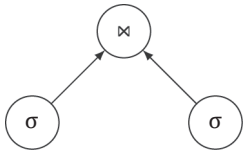


Рис. 1.7 ❖ Межоператорный параллелизм

1.4.4. Масштабируемость

В распределенной среде гораздо проще приспособиться к увеличению размера базы данных и рабочей нагрузки. Обычно систему расширяют путем

добавления в сеть новых процессоров и устройств хранения. Очевидно, что добиться линейного роста «мощности» не всегда возможно, потому что имеются накладные расходы на распределение. Но значительное улучшение все же достижимо. Именно поэтому распределенные СУБД занимают так много места в архитектурах с *горизонтальным* масштабированием в контексте кластерных и облачных вычислений. Под горизонтальным масштабированием понимают добавление новых серверов, не обремененных жесткими связями, что позволяет добиться почти бесконечной масштабируемости. Благодаря простоте добавления новых серверов баз данных распределенная СУБД может поддержать горизонтальное масштабирование.

1.5. Вопросы проектирования

В предыдущем разделе мы обсудили обещания технологии распределенных СУБД, отметив трудности, которые необходимо преодолеть для их реализации. В этом разделе мы продолжим обсуждение, описав проблемы, возникающие при проектировании распределенных СУБД. Этими вопросами мы будем заниматься до конца книги.

1.5.1. Проектирование распределенной базы данных

Вопрос заключается в том, как разместить данные в узлах. Отправной точкой является одна глобальная база данных, а конечным результатом – распределение данных между узлами. Такой подход называется *проектированием сверху вниз*. Есть два основных варианта размещения данных: *секционированное* (или *нереплицированное*) и *реплицированное*. В случае секционированной схемы база данных разбивается на несколько непересекающихся секций, каждая из которых размещается в отдельном узле. Репликация же может быть *полной* (говорят также о *полном дублировании*), когда в каждом узле размещается вся база, или *частичной* (*частичное дублирование*), когда каждая секция хранится в нескольких, но не во всех узлах. Два фундаментальных вопроса проектирования – *фрагментация*, т. е. разбиение базы данных на секции, называемые *фрагментами*, и *распределение*, т. е. оптимальное размещение фрагментов.

С этим связана проблема проектирования системного каталога и управления им. В централизованной СУБД *каталог* содержит метаинформацию о данных (т. е. их описание). В распределенной системе каталог содержит дополнительную информацию о местонахождении данных. Проблемы, связанные с управлением каталогом, по природе своей похожи на проблему размещения базы данных, рассмотренную в предыдущем разделе. Каталог может быть глобальным для всей распределенной СУБД или локальным в каждом узле; он может централизованно храниться в одном узле или распределяться между несколькими узлами; копий каталога может быть одна

или несколько. Проектирование и управление каталогом распределенной базы данных – тема главы 2.

1.5.2. Контроль распределенных данных

Важное требование к СУБД – обеспечение согласованности данных посредством управления доступа к ним. Это называется *контролем данными*, сюда относятся управление представлениями, контроль доступа и гарантии целостности. Распределенность влечет за собой дополнительные проблемы, потому что данные, для которых необходимо проверять правила, находятся в разных узлах, так что требуется распределенная проверка и предоставление гарантий. Эта тема рассматривается в главе 3.

1.5.3. Распределенная обработка запросов

Обработка запросов подразумевает проектирование алгоритмов, которые анализируют запрос и преобразуют его в последовательность операций манипулирования данными. Проблема заключается в нахождении стратегии наиболее эффективного выполнения запроса в сети при заданном определении стоимости. При этом нужно учитывать следующие факторы: распределение данных, затраты на связь, отсутствие достаточной информации, доступной локально. Цель – оптимизировать стоимость при вышеназванных ограничениях, используя внутренне присущий параллелизм для повышения производительности. По своей природе это NP-трудная задача, а подходы к ее решению обычно эвристические. Распределенная обработка запросов подробно обсуждается в главе 4.

1.5.4. Распределенное управление конкурентностью

Под управлением конкурентностью понимается синхронизация доступа к распределенной базе данных, обеспечивающая поддержание целостности данных. Постановки задачи управления конкурентностью в распределенном и централизованном контексте несколько различаются. Думать нужно не только о целостности одной базы данных, но и о согласованности нескольких ее копий. Требование, согласно которому значения нескольких копий каждого элемента данных должны в конечном итоге сходиться к одному и тому же значению, называется *взаимной согласованностью*.

Есть два широких класса решений: *пессимистические*, когда синхронизация выполнения запросов пользователей производится до начала выполнения, и *оптимистические*, когда запросы выполняются, а затем проверяется, не привело ли выполнение к нарушению согласованности. В обоих подходах используются два фундаментальных примитива: *блокировка*, основанная на взаимном исключении операций доступа к элементам данных, и *назначение*

временных меток, позволяющее упорядочить выполнение транзакций на основе присвоенных им временных меток. Существуют различные варианты этих схем, а также гибридные алгоритмы, пытающиеся сочетать оба базовых механизма.

В решениях на основе блокировки возможна взаимоблокировка, поскольку в разных транзакциях могут встречаться взаимно исключающие операции доступа к данным. Хорошо известные методы предотвращения, избегания и обнаружения-восстановления применимы и к распределенным СУБД. Распределенное управление конкурентностью обсуждается в главе 5.

1.5.5. Надежность распределенной СУБД

Выше мы говорили, что одно из потенциальных преимуществ распределенной системы – повышенная надежность и доступность. Но это не происходит автоматически. Важно предоставить механизмы, которые гарантируют согласованность базы данных, а также позволяют обнаруживать ошибки и восстанавливаться после них. Для распределенных СУБД это означает, что при возникновении отказа, когда некоторые узлы становятся неработоспособными или недоступными, базы данных в работающих узлах по-прежнему согласованы и актуальны. А когда вычислительная система или сеть восстанавливается после отказа, распределенная СУБД должна привести базы данных в неработавших узлах в актуальное состояние. Это особенно трудно в случае разделения сети, когда узлы распадаются на две или более групп, между которыми нет связи. Протоколы распределенной надежности – тема главы 5.

1.5.6. Репликация

Если распределенная база данных реплицирована (полностью или частично), то необходимо реализовать протоколы, обеспечивающие согласованность реплик, т. е. все копии одного и того же элемента данных должны иметь одинаковые значения. Эти протоколы могут быть *энергичными* (eager), т. е. обновления принудительно применяются ко всем репликам до завершения транзакции, или *ленивыми* (lazy), т. е. транзакция обновляет только одну копию (называемую *главной*), а потом изменения распространяются на другие копии уже после завершения транзакции. Протоколы репликации обсуждаются в главе 6.

1.5.7. Параллельные СУБД

Как уже было сказано, существует тесная связь между распределенными и параллельными базами данных. Хотя в первом случае предполагается, что каждый узел является отдельным логическим компьютером, на самом деле в большинстве установок узлы представляют собой параллельные кластеры. В этом заключается отмеченное ранее различие между распределением

в пределах одного узла, как в кластерных ЦОДах, и георазпределением. Цели, стоящие перед распределенными СУБД, отличаются от целей параллельных СУБД, основными из которых являются высокая масштабируемость и производительность. В этой книге основное внимание уделено управлению данными в георазпределенных базах данных, но при рассмотрении распределения в пределах одного узла, как в параллельной системе, возникает ряд интересных вопросов управления данными, которые мы обсудим в главе 8.

1.5.8. Интеграция баз данных

Одно из важных направлений развития – движение в сторону «более свободной» федерации источников данных, в т. ч. гетерогенных. В следующем разделе мы увидим, что это привело к развитию систем мультибаз данных (или *федеративных систем баз данных*), которые потребовали пересмотра некоторых фундаментальных методов работы с базами данных. Имеется множество уже распределенных баз данных, а цель – предоставить к ним простой доступ посредством их интеграции (физической или логической). Это требует *проектирования снизу вверх*. Такие системы составляют важную часть современных распределенных сред. В главе 7 мы обсудим системы мультибаз данных, или, как чаще говорят, *интеграцию баз данных*, в т. ч. вопросы проектирования и проблемы, возникающие при обработке запросов.

1.5.9. Альтернативные подходы к распределению

Рост интернета как фундаментальной сетевой платформы поднял важные вопросы, касающиеся предположений, лежащих в основе распределенных систем баз данных. Нам особенно интересны два из них: реинкарнация одноранговых вычислений, с одной стороны, и рост и развитие веба – с другой. В обоих случаях конечная цель – усовершенствование разделения данных, но подходы разнятся, как и возникающие проблемы управления данными. В главе 9 мы обсудим одноранговое управление данными, а в главе 12 – управление веб-данными.

1.5.10. Обработка больших данных и NoSQL

В последние десять лет мы стали свидетелями взрывного роста обработки «больших данных». Точное определение больших данных дать трудно, но, как правило, все согласны с четырьмя характеристиками («четыре V»): данные очень большого объема (*volume*) разнородные (*variety*), обычно поступают с очень высокой скоростью (*velocity*) и могут быть низкокачественными в силу ненадежности источников и внутренних конфликтов (*veracity*). Предприняты значительные усилия для разработки систем работы с «большими данными», поскольку считается, что реляционные СУБД для многих приложений не подходят. Обычно эти усилия принимают одну из двух форм: приверженцы одной линии разрабатывают универсальные вычислительные

платформы (почти всегда горизонтально масштабируемые), а поклонники другой – специальные СУБД, не обладающие реляционными свойствами в полном объеме, но предлагающие более гибкие средства управления данными (так называемые системы NoSQL). Мы обсудим платформы больших данных в главе 10, а системы NoSQL – в главе 11.

1.6. Архитектуры распределенных СУБД

Архитектура системы определяет ее структуру. Это означает, что определены компоненты системы, функции каждой компоненты, взаимосвязи и взаимодействия между компонентами. Для спецификации архитектуры системы необходимо идентифицировать различные модули, их интерфейсы и взаимосвязи в терминах потока данных и управления в системе.

В этом разделе мы опишем четыре «эталонные» архитектуры[1] распределенной СУБД: клиент-серверная, одноранговая, мультибазовая и облачная. Это «идеализированные» взгляды на СУБД в том смысле, что многие коммерческие системы могут от них отклоняться; однако архитектуры служат разумной платформой, на которой можно обсуждать вопросы, относящиеся к распределенным СУБД.

Начнем с обсуждения пространства проектирования, чтобы лучше определить место архитектур, которые будут представлены ниже.

1.6.1. Архитектурные модели для распределенных СУБД

Мы пользуемся классификацией (рис. 1.8) с тремя осями, по которым можно охарактеризовать распределенные СУБД: (1) автономность локальных систем, (2) их распределение и (3) их гетерогенность. Все эти оси независимы друг от друга, и вдоль каждой из них имеется несколько вариантов. Следовательно, пространство проектирования состоит из 18 возможных архитектур. Не все они осмыслены, и большая их часть не имеет отношения к теме этой книги. На рис. 1.8 показаны три архитектуры, на которых мы сосредоточим внимание.

1.6.1.1. Автономность

В этом контексте слово *автономность* относится к распределению управления, а не данных. Речь идет о том, насколько независимо могут работать отдельные СУБД. Автономность зависит от многих факторов, в т. ч. от того, обмениваются ли системы-компоненты (т. е. отдельные СУБД) информацией, могут ли они независимо выполнять транзакции и разрешено ли вносить в них изменения.

---

[1] Эталонная архитектура обычно создается разработчиками стандартов с целью четко определить интерфейсы, подлежащие стандартизации.

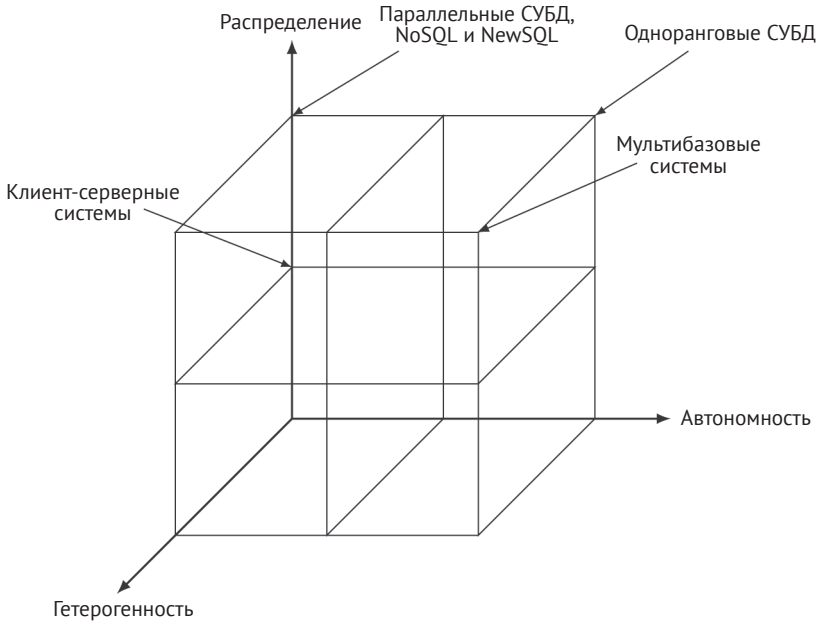


Рис. 1.8 ❖ Альтернативные реализации СУБД

Мы будем пользоваться классификацией, охватывающей важные аспекты указанных характеристик. В этой классификации выделяются три варианта. Первый – *тесная интеграция*, когда каждому пользователю, желающему обращаться к данным, которые могут находиться в нескольких базах, доступен единственный образ всей базы данных. С точки зрения пользователя, данные логически интегрированы в одной базе. В подобных тесно интегрированных системах реализованы диспетчеры данных таким образом, что за обработку каждого запроса пользователя отвечает один диспетчер, даже если запрос обслуживается несколькими. Диспетчеры данных обычно не работают как независимые СУБД, пусть даже в них заложена соответствующая функциональность.

Далее мы выделяем *полуавтономные* системы, состоящие из СУБД, которые могут работать независимо (и обычно так и делают), но приняли решение участвовать в федерации, чтобы предоставить свои локальные данные в общее пользование. Каждая из этих СУБД решает, какие части управляемой ей базы сделать доступными пользователям других СУБД. Они не вполне автономны, т. к. для обмена информацией между собой нуждаются в модификации.

Последний вариант, который мы рассмотрим, – полная изоляция, когда отдельные системы – это самостоятельные СУБД, ничего не знающие ни о существовании других СУБД, ни о том, как с ними взаимодействовать. В таких системах обработка пользовательских транзакций, обращающихся к нескольким базам данных, особенно трудна, потому что нет никаких глобальных средств управления выполнением работы в отдельных СУБД.

1.6.1.2. Распределение

Если автономность относится к распределению (или децентрализации) управления, то ось распределения в нашей классификации имеет отношение к данным. Конечно, мы рассматриваем физическое распределение данных между несколькими узлами; как уже было сказано, пользователь видит данные как один логический пул. Существует много способов распределения СУБД. Мы выделили два больших класса: клиент-серверное и одноранговое (или *полное*) распределение. Вместе с отсутствием распределения получается три архитектуры по этой оси.

В случае клиент-серверного распределения управление данными сосредоточивается на серверах, а клиенты заняты обеспечением среды для приложения, в т. ч. для пользовательского интерфейса. Обязанности поддерживать коммуникацию возлагаются как на клиентские, так и на серверные компьютеры. Клиент-серверные СУБД – это практический компромисс в части функциональности распределения. Существует много способов их структуризации, и каждый из них предлагает разные уровни распределения. Детальное обсуждение мы отложим до раздела 1.6.2.

В одноранговых системах нет различия между клиентскими и серверными машинами. Каждая машина обладает всей функциональностью СУБД и может взаимодействовать с другими машинами для выполнения запросов и транзакций. В самом начале работы над распределенными системами баз данных основное внимание уделялось именно одноранговой архитектуре. Поэтому и в этой книге нас будут интересовать в первую очередь одноранговые (или *полностью распределенные*) системы, хотя многие из рассматриваемых методов переносятся также на клиент-серверные системы.

1.6.1.3. Гетерогенность

Гетерогенность может встречаться в распределенных системах в разных формах – от аппаратной гетерогенности и различий в сетевых протоколах до вариаций диспетчеров данных. С точки зрения этой книги, наиболее важны модели данных, языки запросов и протоколы управления транзакциями. Представление данных с помощью разных средств моделирования уже создает гетерогенность из-за различий в выразительной способности и ограничений разных моделей. Гетерогенность в области языков запросов не только включает использование совершенно разных парадигм доступа к данным в разных моделях (сравните доступ на уровне множеств в реляционных системах и на уровне записей в некоторых объектно-ориентированных системах), но и языковые различия даже в тех случаях, когда в разных системах используется одна и та же модель. Хотя в наши дни SQL стал стандартным реляционным языком запросов, существует много разных реализаций, и в диалекте от каждого производителя есть свои особенности. А на платформах больших данных и систем NoSQL вообще множество существенно различающихся языков и механизмов доступа.

1.6.2. Клиент-серверные системы

Клиент-серверные системы появились на сцене в начале 1990-х годов и оказали огромное влияние на технологию СУБД. Общая идея очень проста и элегантна: отличать функциональность, которую должна предоставлять серверная машина, от функциональности, ожидаемой от клиента. В результате создается *двухуровневая архитектура*, которая позволяет справляться со сложностью современных СУБД и распределения.

В реляционных клиент-серверных СУБД на сервер возлагается большая часть обязанностей по управлению данными. Вся обработка и оптимизация запросов, управление транзакциями и хранением производятся на сервере. Клиент, помимо прикладного и пользовательского интерфейса, включает *клиентский модуль СУБД*, отвечающий за управление данными, кешированными на стороне клиента, и иногда за управление транзакционными блокировками, которые тоже могут кешироваться. На стороне клиента можно также разместить проверку согласованности пользовательских запросов, но это делается нечасто, потому что требует репликации системного каталога на клиентских машинах. Эта архитектура, изображенная на рис. 1.9, типична для реляционных систем, в которых взаимодействие между клиентами и серверами производится на уровне SQL-команд. Иными словами, клиент передает SQL-запросы серверу, не пытаясь ни понять, ни оптимизировать их. Сервер делает большую часть работы и возвращает клиенту результирующее отношение.

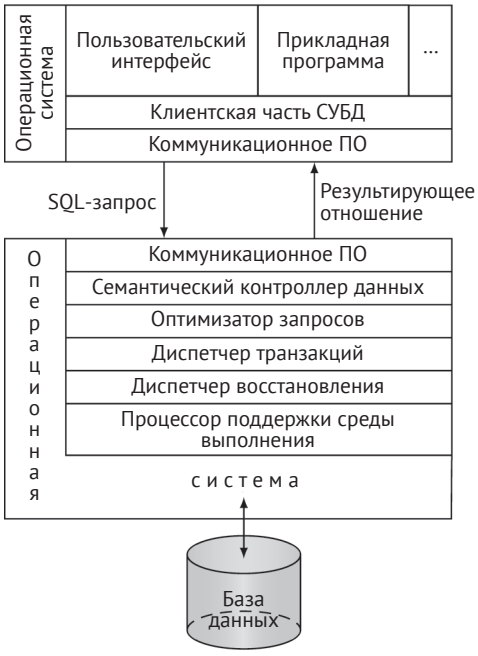


Рис. 1.9 ❖ Эталонная клиент-серверная архитектура

Имеется несколько разных реализаций клиент-серверной архитектуры. Простейший из них – когда существует единственный сервер, к которому обращается несколько клиентов. Этот вариант называется *несколько клиентов – один сервер*. С точки зрения управления данными, это не сильно отличается от централизованных баз данных, поскольку база хранится только на одной машине (сервере) вместе с программным обеспечением для управления ей. Однако существуют важные отличия от централизованных систем в плане выполнения транзакций и управления кешами – поскольку данные кешируются на стороне клиента, необходимо устанавливать протоколы когерентности кешей.

В более сложной клиент-серверной архитектуре имеется несколько серверов (подход *несколько клиентов – несколько серверов*). В этом случае возможны две альтернативные стратегии: либо каждый клиент сам управляет своим соединением с нужным ему сервером, либо клиент знает только о своем «домашнем сервере», который взаимодействует с другими серверами по мере необходимости. При первом подходе упрощается код сервера, но на клиентские машины возлагаются дополнительные обязанности. В результате получаются системы с «тяжелым клиентом». При втором подходе вся функциональность управления данными сосредоточивается на сервере. Таким образом, прозрачность доступа к данным обеспечивается на уровне интерфейса с сервером, а клиенты получаются «легкими».

В системах с несколькими серверами данные секционируются и могут реплицироваться между серверами. Если клиенты легкие, то весь этот механизм прозрачен для них, а серверы могут взаимодействовать друг с другом для ответа на запрос пользователя. Такой подход реализован в параллельных СУБД для повышения производительности посредством параллельной обработки.

Клиент-серверную архитектуру можно расширить, обеспечив более эффективное распределение функций между серверами разного типа: *клиенты* отрабатывают пользовательский интерфейс (например, веб-серверы), *серверы приложений* выполняют прикладные программы, а *серверы баз данных* отвечают за функции управления базой данных. Это приводит к трёхъярусной архитектуре распределенной системы.

Подход на основе серверов приложений (на самом деле *n*-ярусный подход к построению распределенных систем) можно обобщить, введя несколько серверов баз данных и несколько серверов приложений (рис. 1.10), и то же самое можно сделать в классических клиент-серверных архитектурах. В таком случае каждому серверу приложений обычно назначается одно или несколько приложений, а серверы баз данных работают в многосерверном режиме, рассмотренном выше. Кроме того, между приложением и сервером обычно устанавливается балансировщик нагрузки, который маршрутизирует запросы к подходящим серверам.

Подход на основе серверов баз данных, будучи обобщением классической клиент-серверной архитектуры, имеет несколько потенциальных преимуществ. Во-первых, концентрация управления данными в одном месте позволяет разрабатывать специальные методы для повышения надежности и доступности, например с помощью параллелизма. Во-вторых, общую про-

изводительность управления базами данных можно значительно улучшить благодаря тесной интеграции системы баз данных со специализированной для управления базами данных операционной системой. Наконец, серверы баз данных могут пользоваться дополнительным оборудованием, например графическими процессорами и ППВМ, чтобы повысить производительность и доступность данных.

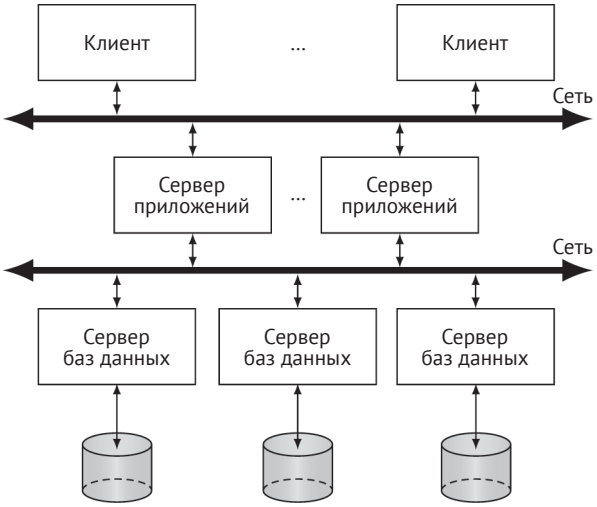


Рис. 1.10 ❖ Серверы распределенной базы данных

Хотя эти преимущества значительны, имеются дополнительные накладные расходы на еще один уровень взаимодействия между серверами приложений и серверами баз данных. Стоимость взаимодействия можно амортизировать, если интерфейс сервера достаточно высокоуровневый и допускает выражение сложных запросов, включающих интенсивную обработку данных.

1.6.3. Одноранговые системы

Все ранние работы по распределенным СУБД были сконцентрированы на одноранговых архитектурах, в которых функциональность всех узлов системы одинакова. У современных одноранговых систем есть два важных отличия от предшественников. Во-первых, это массовое распределение. Если раньше речь шла о нескольких (от силы десяти) узлах, то нынешние системы насчитывают тысячи узлов. Во-вторых, это внутренне присущая всем аспектам узлов гетерогенность и автономность узлов. Хотя, как мы уже обсуждали, это всегда представляло интерес для распределенных баз данных, в сочетании с массовым распределением гетерогенность и автономность узлов приобрели дополнительную важность, исключив из рассмотрения не-

которые подходы. В этой книге мы сначала сосредоточимся на классическом понимании одноранговости (одинаковая функциональность всех узлов), поскольку принципы и фундаментальные методы таких систем очень близки к клиент-серверным системам, а в главе 9 отдельно обсудим современные одноранговые системы.

В этих системах проектирование базы данных ведется сверху вниз, как было описано выше. То есть на входе имеется централизованная база данных со своим определением схемы (*глобальная концептуальная схема* – ГКС). Эта база данных секционируется и размещается в узлах распределенной СУБД. Таким образом, в каждом узле появляется локальная база данных с собственной схемой (называемой *локальной концептуальной схемой* – ЛКС). Пользователь формулирует запросы, ориентируясь на ГКС, независимо от ее местоположения. Распределенная СУБД транслирует глобальные запросы во множество локальных запросов, которые выполняются взаимодействующими между собой компонентами распределенной СУБД в разных узлах. С точки зрения обработки запросов, одноранговые системы и клиент-серверные СУБД предлагают одинаковое представление данных. То есть у пользователя создается впечатление логически единой базы данных, хотя на физическом уровне данные распределены.

Компоненты распределенной СУБД показаны на рис. 1.11. Один компонент отвечает за взаимодействие с пользователями, а другой – за хранение. Первый главный компонент, который мы назвали *процессором пользователей*, состоит из четырех элементов.

1. *Обработчик пользовательского интерфейса* отвечает за интерпретацию команд пользователя и форматирование результатов, отправляемых пользователю.
2. Контроллер данных использует определенные в глобальной концептуальной схеме ограничения целостности и разрешения, для того чтобы проверить, можно ли выполнить запрос пользователя. Эта компонента, которую мы будем подробно изучать в главе 3, также отвечает за авторизацию и другие функции.
3. *Глобальный оптимизатор запросов* определяет стратегию выполнения, минимизирующую функцию стоимости, и преобразует глобальные запросы в локальные, пользуясь для этого глобальной и локальной концептуальными схемами, а также глобальным каталогом. Среди прочего, глобальный оптимизатор запросов отвечает за генерирование наилучшей стратегии выполнения распределенных операций соединения. Эти вопросы обсуждаются в главе 4.
4. *Монитор распределенного выполнения* координирует распределенное выполнение пользовательских запросов. Он также называется *диспетчером распределенных транзакций*. При распределенном выполнении запросов мониторы выполнения, находящиеся в разных узлах, могут взаимодействовать между собой, и обычно так и происходит. Функциональность монитора распределенных транзакций описывается в главе 5.

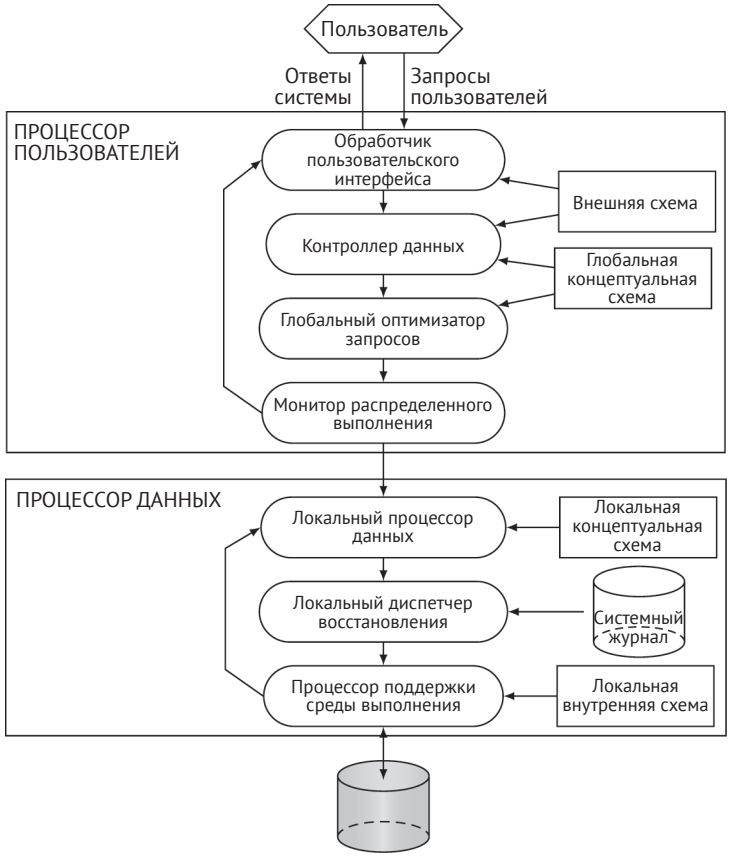


Рис. 1.11 ❖ Компоненты распределенной СУБД

Вторым главным компонентом распределенной СУБД – *процессор данных* – состоит из следующих трех элементов. Все они решаются централизованной СУБД, так что в этой книге мы их рассматривать не будем.

1. *Локальный оптимизатор запросов*, который фактически играет роль селектора пути доступа и отвечает за выбор лучшего пути доступа к элементам данных[1].
2. Локальный диспетчер восстановления отвечает за согласованность локальной базы данных даже после сбоев.
3. Процессор поддержки среды выполнения осуществляет физический доступ к базе данных в соответствии с командами, включенными оптимизатором запросов в план выполнения. Этот процессор реализует интерфейс с операционной системой и содержит диспетчер буфера базы данных (кеша), который отвечает за поддержание буфера в основной памяти и управление доступом к данным.

[1] Термин *путь доступа* относится к структурам данных и алгоритмам, используемым для доступа к данным. Например, типичным путем доступа является индекс, построенный по одному или нескольким атрибутам отношения.

Важно отметить, что наше употребление терминов «процессор пользователей» и «процессор данных» не подразумевает функционального деления, как в клиент-серверных системах. Это деление по преимуществу организационное и не предполагает, что компоненты должны размещаться на разных машинах. В одноранговых системах модули процесса пользователей и процессора данных обычно находятся на каждой машине. Но бывают «узлы запросов», в которых присутствует только процессор пользователей.

1.6.4. Системы управления мультибазами данных

В системах управления мультибазами данных (СУМБД), или мультибазовых системах, отдельные СУБД полностью автономны и понятия кооперации не существует; они могут даже не знать о существовании друг друга и не иметь никаких средств взаимодействия. Естественно, нас интересуют прежде всего распределенные СУМБД, т. е. такие, в которых составляющие СУБД размещены в разных узлах. Многие из рассмотренных нами вопросов являются общими для одноузловых и распределенных СУМБД; в таких случаях мы будем просто говорить «СУМБД» без указания природы. В современной литературе чаще встречается термин *интеграция баз данных*. Мы будем обсуждать такие системы в главе 7. Заметим, однако, что термину «мультибаза» в разных источниках приписывается различный смысл. В этой книге мы будем понимать под ним сказанное выше, но следует помнить, что такая трактовка может не совпадать с встречающейся в литературе.

Различия в уровне автономности СУМБД и распределенных СУБД находят отражение и в их архитектурных моделях. Фундаментальное различие связано с определением глобальной концептуальной схемы. В случае логически интегрированных распределенных СУБД глобальная концептуальная схема определяет концептуальное представление *всей* базы данных, тогда как в случае СУМБД она представляет только набор *некоторых* локальных баз данных, которые локальная СУБД готова предоставить в общее пользование. Отдельные СУБД могут предоставить общий доступ только к части своих данных. Поэтому определения *глобальной базы данных* в СУМБД и распределенных СУБД различаются. Во втором случае глобальная база данных совпадает с объединением локальных, а в первом она является лишь подмножеством (быть может, собственным) этого объединения. В СУМБД глобальная концептуальная схема (иногда называемая *опосредованной схемой*) определяется путем интеграции локальных концептуальных схем (или их частей).

Компонентная архитектурная модель распределенной СУМБД существенно отличается от модели распределенной СУБД тем, что каждый узел полноценной СУБД управляет своей базой данных. СУМБД предоставляет слой программного обеспечения, работающий поверх этих отдельных СУБД и предлагающий пользователям средства доступа к разным базам данных (рис. 1.12). Отметим, что в распределенной СУМБД слой СУМБД может работать в нескольких узлах или находиться в одном узле, который и предоставляет соответствующие службы. Также заметим, что, с точки зрения отдель-

ных СУБД, слой СУМБД является просто еще одним приложением, которое отправляет запросы и получает ответы.

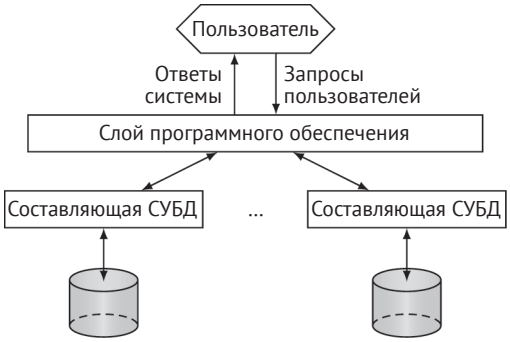


Рис. 1.12 ❖ Компоненты СУМБД

Популярной реализацией архитектуры СУМБД является подход на основе посредников и оберток (рис. 1.13). *Посредником* называется «программный модуль, который использует закодированные знания о некоторых множествах или подмножествах данных для порождения информации, предназначенной более высоким уровням приложений» [Wiederhold 1992]. Таким образом, каждый посредник выполняет конкретную функцию с четко определенным интерфейсом. При такой архитектуре каждый модуль, принадлежащий слою СУМБД на рис. 1.12, реализован в виде посредника. Поскольку посредники могут быть надстроены над другими посредниками, можно организовать многоуровневую реализацию. Уровень посредников реализует ГКС. Именно на этом уровне обрабатываются пользовательские запросы к ГКС и предоставляется функциональность СУМБД.

Посредники обычно работают с общей моделью данных и языком определения интерфейса. Чтобы справиться с потенциальной гетерогенностью СУБД-источников, создаются *обертки*, задача которых – предоставить отображение между СУБД-источником и посредником. Например, если СУБД-источник реляционный, а посредник объектно-ориентированный, то необходимые отображения реализуются обертками. Точная роль и функция посредников зависят от реализации. В некоторых случаях посредники не делают ничего, кроме трансляции, тогда они называются «тонкими». Но иногда посредники берут на себя часть функциональности обработки запросов.

Набор посредников можно рассматривать как промежуточный уровень, предоставляющий службы, надстроенные над системами-источниками. ПО промежуточного уровня – тема, которая активно разрабатывалась в прошлом десятилетии, когда были созданы весьма развитые системы, предлагающие продвинутые средства для разработки распределенных приложений. Обсуждаемые нами посредники – лишь часть функциональности таких систем.

Рис. 1.13 ❖ Архитектура с посредниками/обертками

1.6.5. Облачные вычисления

Облачные вычисления привели к тектоническим сдвигам в методах развертывания масштабируемых приложений пользователями и организациями, а в особенности приложений для управления данными. Видится система (обычно она изображается в виде облака), предоставляющая через интернет по запросу надежные службы с простым доступом к практически бесконечным ресурсам – вычислительным, хранения и сетевым. Пользуясь очень простыми веб-интерфейсами и за скромную плату, пользователи могут выносить такие сложные задачи, как хранение данных, управление базами данных, администрирование системы или развертывание приложений за пределы предприятия, в очень крупные центры обработки данных, эксплуатируемые поставщиками облачных служб. Таким образом, сложность управления программно-аппаратной инфраструктурой переходит от пользовательской организации к облачному поставщику.

Облачные вычисления – это естественная эволюция и сочетание различных моделей вычислений, предложенных для поддержки приложений через веб: сервисно-ориентированных архитектур (SOA) для высокоуровневого взаимодействия приложений с помощью веб-служб, служебных вычислений для упаковки вычислительных ресурсов и ресурсов хранения в виде служб, кластерных технологий и виртуализации для управления многочисленными ресурсами (вычислительными и хранения) и автономных вычислений для самоконтроля сложной инфраструктуры. Облако предоставляет различные уровни функциональности:

- инфраструктура как услуга (IaaS): предложение вычислительной инфраструктуры (вычисления, сеть и ресурсы хранения) в виде услуги;
- платформа как услуга (PaaS): предложение вычислительной платформы вместе с инструментами разработки и API в виде услуги;
- программное обеспечение как услуга (SaaS): предложение прикладного ПО в виде услуги;
- база данных как услуга (DaaS): предложение базы данных в виде услуги.

Уникальность облачных вычислений заключается в предоставлении таких комбинаций услуг, которые наилучшим образом отвечают требованиям пользователя. С технической точки зрения, проблема в том, чтобы сравнительно дешево поддержать очень большую инфраструктуру, способную управлять множеством пользователей и ресурсов, обеспечивая высокое качество обслуживания.

Дать точное определение облачных вычислений, с которым все были бы согласны, трудно, т. к. точек зрения много (коммерческая, маркетинговая, техническая, исследовательская и т. д.). Но можно принять рабочее определение – «облако по запросу предоставляет через интернет ресурсы и услуги, обычно сравнимые по масштабу и надежности с центром обработки данных» [Grossman and Gu 2009]. В этом определении отражена главная цель (предоставление ресурсов и услуг по запросу через интернет) и основные требования к технической поддержке (сравнимые по масштабу и надежности с центром обработки данных). Поскольку доступ к ресурсам осуществляется посредством служб, то пользователь получает все запрошенное в виде услуг. Поэтому, как и в сфере услуг, облачные поставщики могут работать по модели ценообразования «плата по факту», когда пользователь платит только за потребленные ресурсы.

Основными функциями, предоставляемыми облаками, являются: безопасность, управление каталогом, управление ресурсами (подготовка, выделение, мониторинг) и управление данными (хранение, управление файлами, управление базой данных, репликация данных). Кроме того, облако реализует начисление платы, бухгалтерский учет и гарантии соглашения об уровне обслуживания (SLA). Перечислим типичные преимущества облачных вычислений.

- **Стоимость.** Затраты заказчика заметно сокращаются, поскольку ему не нужно нести расходы на владение и управление инфраструктурой; счета выставляются только за потребленные ресурсы. С точки зрения облачного поставщика, поддержание консолидированной инфраструктуры и отнесение затрат на многих заказчиков уменьшает стоимость владения и эксплуатации.
- **Простота доступа и использования.** Облако скрывает сложность ИТ-инфраструктуры и обеспечивает прозрачность местоположения и распределения. Поэтому заказчик может получить доступ к ИТ-услугам в любое время и из любого места, где можно подключиться к интернету.
- **Качество обслуживания.** Эксплуатация ИТ-инфраструктуры специализированным поставщиком, который имеет обширный опыт работы с очень крупной инфраструктурой (в т. ч. своей собственной), повышает качество обслуживания и операционную эффективность.

- **Инновационность.** Использование самых свежих инструментов и приложений, предлагаемых облаком, способствует внедрению современных технологий, повышая тем самым способность заказчиков к восприятию инноваций.
- **Эластичность.** Способность к динамическому вертикальному масштабированию (вверх и вниз) для адаптации к изменяющимся условиям – важное преимущество. Обычно это достигается посредством виртуализации серверов – технологии, которая позволяет нескольким приложениям работать в виртуальных машинах (ВМ) на одном физическом компьютере, т. е. как если бы они работали на физически разных компьютерах. Тогда заказчик может затребовать вычислительные экземпляры в виде ВМ и присоединять к ним ресурсы хранения по мере необходимости.

Однако имеются и недостатки, которые нужно хорошо понимать, принимая решение о переходе в облако. Они такие же, как при передаче приложений и данных в аутсорсинг внешней компании.

- **Зависимость от поставщика.** Заказчик, как правило, оказывается прочно привязан к облачному поставщику, поскольку использует его закрытое программное обеспечение и форматы, а стоимость передачи данных за пределы системы поставщика высока. Все это сильно затрудняет миграцию из облачной службы.
- **Потеря контроля.** Заказчик может потерять административный контроль над критическими операциями, в т. ч. временем простоя системы, например для перехода на новую версию ПО.
- **Безопасность.** Поскольку доступ к облачным данным возможен из любого места, где есть интернет, атаки могут скомпрометировать данные компании. Безопасность облака можно повысить с помощью дополнительных средств, например путем организации виртуального частного облака, но иногда их трудно интегрировать с политикой безопасности компании.
- **Скрытые затраты.** Модификация приложения для работы в облаке типа SaaS или PaaS может потребовать весьма дорогостоящих доработок.

Не существует стандарта облачной архитектуры, и, по-видимому, он никогда не появится, потому что разные облачные поставщики предлагают разные облачные службы (IaaS, PaaS, SaaS и т. д.) разными способами (публичное, частное, виртуальное частное и т. д.), зависящими от бизнес-модели. Поэтому мы обсудим упрощенную облачную архитектуру с упором на управление базами данных.

Как правило, облако включает несколько территориально разнесенных узлов, или ЦОДов (рис. 1.14), каждый со своими ресурсами и данными. Крупные облачные поставщики разделяют весь мир на несколько регионов, в каждом из которых находится несколько узлов. Тому есть три основные причины. Во-первых, задержка в регионе пользователя низкая, потому что запросы поступают ближайшему узлу. Во-вторых, репликация данных между узлами в разных регионах обеспечивает высокую доступность и, в частности, устойчивость к катастрофическим отказам узлов. В-третьих, законы некоторых стран о защите персональных данных пользователей вынуждают облачных поставщиков располагать ЦОДы в соответствующем регионе (например, в Ев-

ропе). Многоузловая прозрачность обычно является режимом по умолчанию, т. е. облако представляется «централизованным», а поставщик услуг может оптимизировать выделение ресурсов пользователям. Однако некоторые облачные поставщики (в частности, Amazon и Microsoft) делают узлы видимыми пользователям (или разработчикам приложений). Это позволяет либо выбрать конкретный ЦОД для установки приложения вместе с его базой данных, либо развернуть очень большое приложение в нескольких узлах, взаимодействующих посредством веб-служб. Например, глядя на рис. 1.14, можно представить, что Клиент 1 сначала подключается к приложению в ЦОДе 1, которое затем обращается к приложению в ЦОДе 2 через веб-службу.

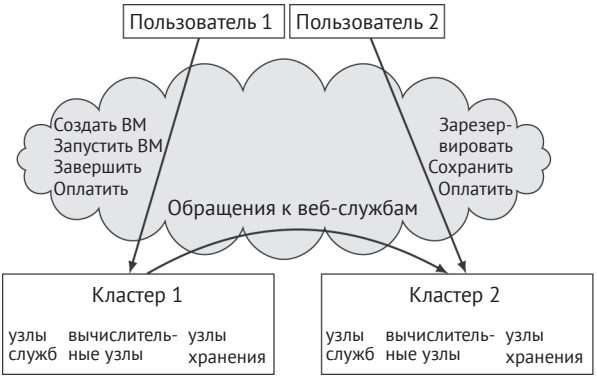


Рис. 1.14 ❖ Упрощенная архитектура облака

Архитектура облачного узла (центра обработки данных) обычно трехъярусная. На первом ярусе располагаются веб-клиенты, которые обращаются к облачным веб-серверам, как правило, пользуясь маршрутизатором или балансировщиком нагрузки в облачном узле. На втором ярусе располагаются веб-серверы и серверы приложений, которые работают с клиентами и предоставляют бизнес-логику. Третий уровень включает серверы баз данных. Возможны и серверы других типов, например для кеширования данных между серверами приложений и баз данных. Таким образом, облачная архитектура предлагает два уровня распределения: территориальное распределение между узлами на основе глобальной сети и распределение между серверами на одном узле, обычно образующими кластер компьютеров. На первом уровне применяются методы территориально распределенных СУБД, а на втором – методы параллельных СУБД.

Облачные вычисления изначально были спроектированы веб-гигантами для выполнения своих приложений очень крупного масштаба в ЦОДах, насчитывающих тысячи серверов. Системы обработки больших данных (глава 10) и системы NoSQL/NewSQL (глава 11) специально «заточены» под требования таких приложений в облаке и используют методы управления распределенными данными. С появлением решений типа SaaS и PaaS облачным поставщиком также понадобилось предоставлять очень большому числу заказчиков, называемых *арендаторами*, небольшие приложения, каждое из

которых ведет собственную (небольшую) базу данных, к которой обращаются его пользователи. Выделять отдельный сервер каждому арендатору слишком расточительно в плане аппаратных ресурсов. Чтобы уменьшить непроизводительное расходование ресурсов и снизить операционные затраты, облачные поставщики обычно организуют разделение ресурсов между арендаторами, применяя «многоарендные» архитектуры, в которых один сервер может поддерживать несколько арендаторов. В различных моделях многоарендности приняты разные компромиссы между производительностью, изоляцией (в плане безопасности и производительности) и сложностью проектирования. В IaaS применяется простая модель совместного использования оборудования, которая обычно реализуется с помощью виртуализации серверов, когда под каждую арендуемую базу данных и операционную систему отводится своя ВМ. Эта модель обеспечивает высокую изоляцию с точки зрения безопасности. Но степень совместного использования ресурсов ограничена из-за наличия избыточных экземпляров СУБД (по одному на ВМ), которые не взаимодействуют между собой и управляют ресурсами независимо. В контексте SaaS, PaaS или DaaS можно выделить три основные модели многоарендного управления базами данных, в которых степень разделения ресурсов и производительности увеличивается, но за счет меньшей изоляции и большей сложности.

- **Разделяемый сервер СУБД.** В этой модели арендаторы разделяют сервер с общим экземпляром СУБД, но база данных у каждого арендатора своя. Большинство СУБД предлагают поддержку нескольких баз данных одним экземпляром сервера. Поэтому такую модель легко поддержать. Она обеспечивает строгую изоляцию на уровне базы данных и более эффективна, чем совместное использование оборудования, поскольку экземпляр СУБД осуществляет полный контроль аппаратных ресурсов. Однако раздельное управление каждой базой данных все же ведет к неэффективному управлению ресурсами.
- **Разделяемая база данных.** В этой модели арендаторы разделяют одну базу данных, но у каждого имеется своя схема и набор таблиц. Консолидация баз данных обычно обеспечивается дополнительным уровнем абстракции внутри СУБД. Эта модель реализована в некоторых СУБД (например, Oracle) с помощью одной контейнерной базы, содержащей несколько баз данных. Она предлагает эффективное использование ресурсов и хорошую изоляцию на уровне схемы. Однако при наличии большого числа (тысяч) арендаторов сервера образуется много мелких таблиц, что ведет к заметным накладным расходам.
- **Разделяемые таблицы.** В этой модели арендаторы разделяют базу данных, схему и таблицы. Чтобы понять, какому арендатору принадлежит конкретная строка таблицы, обычно заводят дополнительный столбец tenant_id. Хотя при этом достигается лучшая степень разделения ресурсов (например, кеша в памяти), изоляция ниже – как в плане безопасности, так и в плане производительности. Например, крупным заказчикам будет принадлежать больше строк в разделяемых таблицах, из-за чего страдает производительность заказчиков поменьше.

1.7. Библиографические замечания

По распределенным СУБД не так много книг. Две из числа первых, Ceri and [Pelagatti 1983] и [Bell and Grimson 1992], больше не издаются. В более поздней книге [Rahimi and Haug 2010] изложены некоторые классические вопросы, затрагиваемые и в этой книге. Кроме того, почти в каждой книге по базам данных есть глава, посвященная распределенным СУБД.

Первопроходческие системы Distributed INGRES и SDD-1 обсуждаются в работах [Stonebraker and Neuhold 1977] и [Wong 1977] соответственно.

Введение в проектирование баз данных имеется в работе [Levin and Morgan 1975], а более полно рассматривается в работе [Ceri et al. 1987]. Обзор алгоритмов распределенного хранения файлов см. в статье [Dowdy and Foster 1982]. Управление каталогами не получило подробного освещения в научном сообществе, но об общих методах можно прочитать в работах [Chu and Nahouraii 1975] и [Chu 1976]. Обзор методов обработки запросов см. в работе [Sacco and Yao 1982]. Описание алгоритмов управления конкурентностью приведено в работах [Bernstein and Goodman 1981] и [Bernstein et al. 1987]. Управление взаимоблокировками также является темой обширных исследований; введение имеется в статье [Isloor and Marsland 1980], но чаще цитируется работа [Obermack 1982]. Хорошими обзорами по методам обнаружения взаимоблокировок являются работы [Knapp 1987] и [Elmagarmid 1986]. Надежность относится к числу вопросов, обсуждаемых в работе [Gray 1979], одной из основополагающих в этой области. Из других важных работ на эту тему упомянем [Verhofstadt 1978] и [Härder and Reuter 1983]. Работа [Gray 1979] также является первой, в которой обсуждаются вопросы поддержки распределенных баз данных со стороны операционной системы; та же тема рассматривается в работе [Stonebraker 1981]. К сожалению, в обеих работах на первом плане стоят централизованные базы данных. Очень хороший ранний обзор систем управления мультибазами данных см. в работе [Sheth and Larson 1990], а в статье [Wiederhold 1992] предложен подход к СУМБД на основе посредников и оберток. Облачные вычисления – тема огромного количества недавно вышедших книг; в качестве хорошей отправной точки назовем книгу [Agrawal et al. 2012], а в работе [Cusumano 2010] приведен неплохой краткий обзор. Архитектура, рассмотренная в разделе 1.6.5, заимствована из работы [Agrawal et al. 2012]. Различные многоарендные модели в облачных средах обсуждаются в работах [Curino et al. 2011] и [Agrawal et al. 2012].

Существует немало предложений по выбору архитектуры. Из наиболее интересных назовем работу [Schreiber 1977], в которой подробно описано расширение системы ANSI/SPARC как попытка включить гетерогенность моделей данных, а также предложение [Mohan and Yeh 1978]. Понятно, что они относятся к самому началу развития технологии распределенных СУБД. Детальная компонентная архитектура системы, приведенная на рис. 1.11, заимствована из работы [Rahimi 1987]. Альтернатива классификации, показанной на рис. 1.8, приведена в работе [Sheth and Larson 1990].

В книге [Agrawal et al. 2012] великолепно изложены проблемы и концепции управления данными в облаке, в т. ч. распределенные транзакции, системы обработки больших данных и многоарендные базы данных.

Глава 2

Проектирование распределенных и параллельных баз данных

Проектирование типичной базы данных – процесс, который начинается со сбора требований и заканчивается определением схемы, описывающей множество отношений. Проектирование распределения начинается с этой глобальной концептуальной схемы (ГКС) и сводится к двум задачам: *секционирование* (*фрагментация*) и *размещение*. В одних методах обе задачи объединяются в единый алгоритм, в других реализуются порознь, как показано на рис. 2.1. Обычно в этом процессе используется вспомогательная информация, которая также изображена на рисунке, хотя часть ее необязательна (отсюда и штриховые линии).

Причины и цели фрагментации в распределенных и параллельных СУБД несколько различаются. Первая причина – *локальность данных*. По возможности мы хотели бы, чтобы при выполнении запросов производился доступ к данным только в одном узле, чтобы избежать дорогостоящего доступа к удаленным данным. Вторая важная причина заключается в том, что фрагментация позволяет одновременно выполнять несколько запросов (благодаря *межзапросному параллелизму*). Фрагментация отношений также приводит к параллельному выполнению одного запроса посредством разделения его на подзапросы, оперирующие фрагментами; это называется *внутризапросным параллелизмом*. Поэтому в распределенной СУБД фрагментация потенциально может уменьшить количество дорогих операций доступа к удаленным данным и повысить степень межзапросного и внутризапросного параллелизма.

В параллельной СУБД локализация данных менее важна, потому что затраты на коммуникацию между узлами гораздо меньше, чем в гораспределенных СУБД. Гораздо важнее балансировка нагрузки, поскольку мы хотим, чтобы все узлы выполняли примерно одинаковый объем работы. В противном случае появляется опасность пробуксовки всей системы, когда один или

несколько узлов перегружены, а остальные простаивают. При этом также возрастает задержка при выполнении запросов и транзакций, т. к. приходится ждать завершения операций на перегруженных узлах. Важны как внутризапросный, так и межзапросный параллелизм (мы будем обсуждать это в главе 8), но в некоторых современных системах обработки больших данных (глава 10) межзапросному параллелизму уделяется больше внимания.

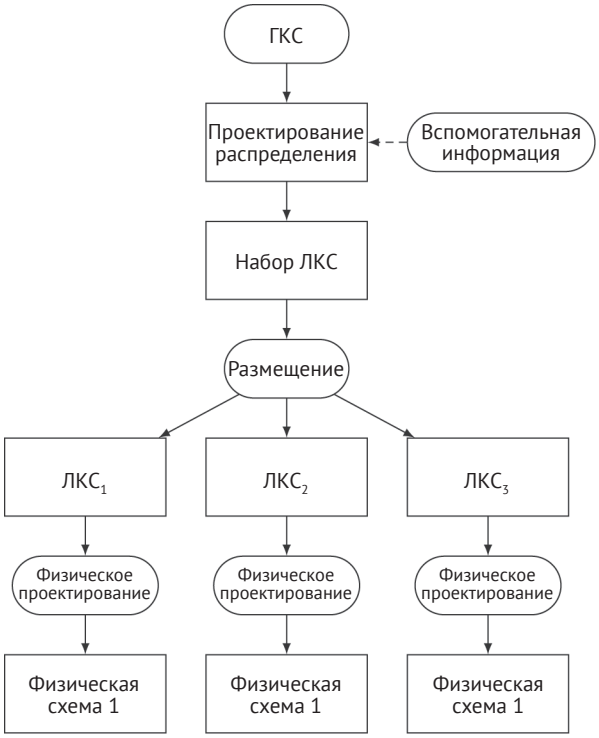


Рис. 2.1 ❖ Процесс проектирования распределения

Фрагментация важна для производительности системы, но она же вызывает трудности в распределенных СУБД. Не всегда возможно полностью локализовать запросы и транзакции, так чтобы осуществлять доступ к данным только в одном узле, – в таком случае говорят о *распределенных запросах* и *распределенных транзакциях*. Из-за них производительность обработки снижается, поскольку, например, приходится нести затраты на выполнение распределенных соединений и на фиксацию распределенных транзакций (см. главу 5). Один из способов решить эту проблему для запросов чтения – реплицировать данные на несколько узлов (см. главу 6), но это лишь повышает накладные расходы, связанные с распределенными транзакциями. Вторая проблема касается семантического управления данными, точнее проверкой целостности. В результате фрагментации атрибуты, участвующие

в ограничении (см. главу 3), могут попасть в разные фрагменты, для которых выделено место в разных узлах. В таком случае сама проверка ограничений целостности включает распределенное выполнение, что обходится дорого. Вопрос о распределенном контроле данных мы рассмотрим в следующей главе. Таким образом, проблема состоит в том, чтобы секционировать[1] данные и выделить для них место, так чтобы большинство пользовательских запросов и транзакций ограничивались одним узлом, сведя тем самым к минимуму количество распределенных запросов и транзакций.

Обсуждение в этой главе будет построено, следуя рис. 2.1: сначала мы обсудим фрагментацию глобальной базы данных (раздел. 2.1), а затем – как разместить фрагменты в узлах распределенной базы данных (раздел 2.2). В этой методике единицей распределения и размещения является фрагмент. Существуют также подходы, в которых фрагментация и размещение объединены, мы обсудим их в разделе 2.3. Наконец, в разделе 2.4 мы рассмотрим методы, адаптирующиеся к изменениям в базе данных и рабочей нагрузке.

В этой главе, как и во всей книге, мы будем использовать базу данных конструкторской компании, представленную в предыдущей главе. На рис. 2.2 показан экземпляр этой базы.

EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

EMP

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E3	P3	Consultant	10
E3	P4	Engineer	48
E4	P2	Programmer	18
E5	P2	Manager	24
E6	P4	Manager	48
E7	P3	Engineer	36
E8	P3	Manager	40

PROJ

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris

PAY

TITLE	SAL
Elect. Eng.	40000
Syst. Anal.	34000
Mech. Eng.	27000
Programmer	24000

Рис. 2.2 ❖ Пример базы данных

[1] Есть один мелкий терминологический нюанс, связанный с употреблением слов «фрагментация» и «секционирование»: в распределенных СУБД чаще говорят о фрагментации, а в параллельных предпочитают термин «секционирование». Мы не отдаем предпочтения ни тому, ни другому термину, употребляя их в этой главе и во всей книге как синонимы.

2.1. Фрагментация данных

Реляционные таблицы можно секционировать *горизонтально* или *вертикально*. Основой горизонтальной фрагментации служит оператор выборки, предикат которого определяет способ фрагментации, тогда как вертикальная фрагментация осуществляется с помощью оператора проецирования. Разумеется, фрагментация может быть вложенной. Если на разных уровнях вложенности применяется фрагментация разного типа, то мы говорим о *гибридной фрагментации*.

Пример 2.1. На рис. 2.3 отношение PROJ, показанное на рис. 2.2, разбито по горизонтали на два фрагмента: $PROJ_1$ содержит информацию о проектах с бюджетом меньше 200 000 долларов, а $PROJ_2$ – о проектах с большим бюджетом.

$PROJ_1$

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York

$PROJ_2$

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	255000	New York
P4	Maintenance	310000	Paris

Рис. 2.3 ❖ Пример горизонтального секционирования

Пример 2.2. На рис. 2.4 то же отношение PROJ секционировано по вертикали на два фрагмента: $PROJ_1$ и $PROJ_2$. Фрагмент $PROJ_1$ содержит только информацию о бюджетах проектов, а $PROJ_2$ – информацию об их названиях и местах выполнения. Важно, что первичный ключ отношения (PNO) включен в оба фрагмента. ♦

$PROJ_1$

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000

$PROJ_2$

PNO	PNAME	LOC
P1	Instrumentation	Montreal
P2	Database Develop.	New York
P3	CAD/CAM	New York
P4	Maintenance	Paris

Рис. 2.4 ❖ Пример вертикального секционирования

Горизонтальная фрагментация встречается чаще, особенно в параллельных СУБД (в этом случае в литературе нередко употребляется термин

шардинг). Причина связана с *внутризапросным параллелизмом*[1], за который ратует большинство платформ больших данных. Однако вертикальная фрагментация нашла успешное применение в таких столбцовых параллельных СУБД, как MonetDB и Vertica, предназначенных для аналитических приложений, где необходим быстрый доступ к некоторым атрибутам.

Систематические приемы фрагментации, обсуждаемые в этой главе, гарантируют, что в процессе фрагментации не произойдет никаких изменений в семантике базы данных, скажем потери части данных. Поэтому необходимо сказать несколько слов о *полноте* и *реконструируемости*. В случае горизонтальной фрагментации желательным свойством может быть также *дизъюнктность* фрагментов (если только мы специально не хотим реплицировать отдельные кортежи, о чем будет сказано ниже).

1. *Полнота*. Если отношение R разложено на фрагменты $F_R = \{R_1, R_2, \ldots, R_n\}$, то каждый элемент данных, присутствующий в R, может быть найден в одном или нескольких фрагментах R_i. Это свойство, эквивалентное *беспотерьной декомпозиции* нормализации, важно и при фрагментации, потому что гарантируется, что данные глобального отношения отображаются на фрагменты без потерь. Заметим, что в случае горизонтальной фрагментации «элемент» обычно обозначает кортеж, а в случае вертикальной – атрибут.
2. *Реконструкция*. Если отношение R разложено на фрагменты $F_R = \{R_1, R_2, \ldots, R_n\}$, то должна существовать возможность определить реляционный оператор ∇ такой, что

   $$R = \nabla R_i, \qquad \forall R_i \in F_R.$$

   Оператор ∇ будет разным для разных форм фрагментации; важно, однако, что его можно подобрать. Возможность реконструкции отношения по его фрагментам гарантирует, что ограничения, определенные в виде зависимостей, сохраняются.
3. *Дизъюнктность*. Если отношение R разложено по горизонтали на фрагменты $F_R = \{R_1, R_2, \ldots, R_n\}$ и элемент данных d_i принадлежит R_j, то он не принадлежит никакому другому фрагменту R_k ($k \neq j$). Это условие гарантирует, что горизонтальные фрагменты дизъюнктны. Если отношение R разложено по вертикали, то атрибуты, определяющие первичный ключ, обычно повторяются во всех фрагментах (для реконструкции). Поэтому в случае фрагментации по вертикали дизъюнктность требуется только для атрибутов, не входящих в состав первичного ключа отношения.

---

[1] В этой главе термины «запрос» и «транзакция» употребляются как синонимы, поскольку оба относятся к рабочей нагрузке на систему, каковая играет первостепенную роль при проектировании распределения. Как отмечено в главе 1 и будет подробно обсуждаться в главе 5, транзакции дают дополнительные гарантии, поэтому накладные расходы на них выше, и в нужных местах обсуждения мы будем это учитывать.

2.1.1. Горизонтальная фрагментация

Как было сказано выше, горизонтальная фрагментация разбивает отношение вдоль кортежей, т. е. каждый фрагмент содержит некоторое подмножество кортежей отношения. Существует два варианта горизонтальной фрагментации: главная и производная. *Главная горизонтальная фрагментация* отношения производится с помощью предикатов, определенных на данном отношении. С другой стороны, *производная горизонтальная фрагментация* – это фрагментация отношения, получающаяся с помощью предикатов, определенных на другом отношении.

Ниже в этом разделе мы рассмотрим алгоритм выполнения обеих фрагментаций. Но сначала поговорим о том, какая информация необходима для горизонтальной фрагментации.

2.1.1.1. Требования к дополнительной информации

Необходимая информация о базе данных относится к глобальной концептуальной схеме, прежде всего к тому, как отношения связаны между собой, особенно с помощью соединений. Один из способов собрать такую информацию – явным образом смоделировать связи между первичным и внешним ключами в виде *графа соединений*. В этом графе каждое отношение R_i представлено вершиной, а R_i связано с R_j ориентированным ребром L_k, если существует эквисоединение R_i с R_j, описываемое парой первичный ключ – внешний ключ.

Пример 2.3. На рис. 2.5 показаны ребра, связывающие отношения в базе данных, изображенной на рис. 2.2. Заметим, что направление ребра определяет связь один-ко-многим. Например, для каждой должности существует много работников, занимающих такую должность, поэтому между отношениями PAY и EMP проведено ребро. Аналогично связь многие-ко-многим между отношениями EMP и PROJ выражается с помощью двух ребер, ведущих к отношению ASG. ♦

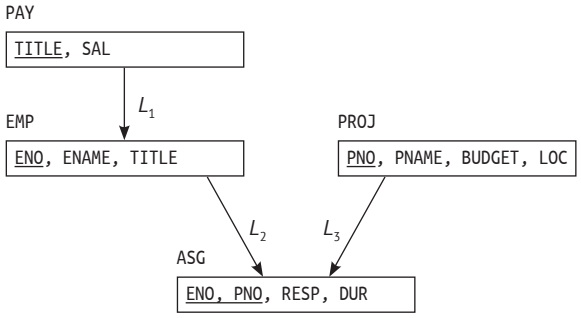


Рис. 2.5 ❖ Граф соединений, представляющий связи между отношениями

Отношение на конце ребра называется *источником* ребра, а отношение в его начале – *целью*. Определим две функции, *source* и *target*, отображаю-

щие множество ребер в множество отношений. Для ребра L_1 на рис. 2.5 *source*(L_1) = PAY, *target*(L_1) = EMP.

Для горизонтальной фрагментации полезно также знать мощность каждого отношения R, обозначаемую *card*(R).

В этих подходах используется информация о рабочей нагрузке, т. е. о запросах, предъявляемых к базе данных. Особенно важны предикаты, встречающиеся в пользовательских запросах. Во многих случаях полностью проанализировать рабочую нагрузку невозможно, поэтому проектировщик сосредоточивается на самых важных запросах. В информатике хорошо известно эвристическое правило «80/20», применимое и в этом случае: 80 % всех операций доступа приходится на 20 % самых частых запросов, поэтому выявления этих 20 % обычно достаточно для получения фрагментации, которая улучшает большинство операций доступа к распределенной базе данных.

В этот момент нас интересует определение *простых предикатов*. Если задано отношение R(A_1, A_2, …, A_n), где A_i – атрибут с областью определения D_i, то простой предикат p_j, определенный на R, имеет вид

p_j : A_i θ *Value*,

где θ ∈ {=, <, ≠, ⩽, >, ⩾}, а *Value* выбирается из области определения A_i (*Value* ∈ D_i). Будем обозначать Pr_i множество всех простых предикатов, определенных на отношении R_i. Элементы Pr_i обозначаются p_{ij}.

Пример 2.4. Для отношения PROJ на рис. 2.2

PNAME = "Maintenance" и BUDGET ⩽ 200000

являются простыми предикатами. ♦

В пользовательских запросах часто встречаются и более сложные предикаты, представляющие собой булевы комбинации простых, например *элементарная конъюнкция* (минтерм), т. е. конъюнкция простых предикатов. Поскольку любое булево выражение можно представить в виде конъюнктивной нормальной формы, использование элементарных конъюнкций при проектировании алгоритмов не приводит к потере общности.

Если задано множество простых предикатов $Pr_i = \{p_{i1}, p_{i2}, \ldots, p_{im}\}$ для отношения R_i, то множество элементарных конъюнкций $M_i = \{m_{i1}, m_{i2}, \ldots, m_{iz}\}$ определяется как

$$M_i = \left\{ m_{ij} = \bigwedge_{p_{ik} \in Pr_i} p_{ik}^* \right\}, \quad 1 \le k \le m, \; 1 \le j \le z,$$

где $p_{ik}^* = p_{ik}$ или $p_{ik}^* = \neg p_{ik}$. Таким образом, каждый простой предикат может входить в элементарную конъюнкцию в естественном виде или в виде отрицания.

Что такое отрицание предиката, понятно для предикатов равенства вида *Атрибут* = *Значение*. Для предикатов неравенства отрицание следует рассматривать как дополнение. Например, отрицанием простого предиката *Атрибут* ⩽ *Значение* является *Атрибут* > *Значение*. С нахождением дополнения в бесконечных множествах возникают теоретические проблемы, и, кроме

того, есть практическая проблема – иногда дополнение трудно определить. Так, если имеются два простых предиката *Нижняя_граница* ⩽ *Атрибут_1* и *Атрибут_1* ⩽ *Верхняя_граница*, то их дополнениями являются ¬(*Нижняя_граница* ⩽ *Атрибут_1*) и ¬(*Атрибут_1* ⩽ *Верхняя_граница*). Однако два исходных предиката можно записать в виде предиката *Нижняя_граница* ⩽ *Атрибут_1* ⩽ *Верхняя_граница*, дополнение к которому ¬(*Нижняя_граница* ⩽ *Атрибут_1* ⩽ *Верхняя_граница*) определить не так-то просто. Поэтому мы ограничимся простыми предикатами.

Пример 2.5. Рассмотрим отношение PAY на рис. 2.2. Ниже перечислено несколько предикатов, которые можно определить над PAY.

p_1 : TITLE = "Elect. Eng."
p_2 : TITLE = "Syst. Anal."
p_3 : TITLE = "Mech. Eng."
p_4 : TITLE = "Programmer"
p_5 : SAL ⩽ 30000

А вот несколько элементарных конъюнкций, которые можно определить над этими простыми предикатами.

m_1 : TITLE = "Elect. Eng." ∧ SAL ⩽ 30000
m_2 : TITLE = "Elect. Eng." ∧ SAL > 30000
m_3 : ¬(TITLE = "Elect. Eng.") ∧ SAL ⩽ 30000
m_4 : ¬(TITLE = "Elect. Eng.") ∧ SAL > 30000
m_5 : TITLE = "Programmer" ∧ SAL ⩽ 30000
m_6 : TITLE = "Programmer" ∧ SAL > 30000 ♦

Это всего лишь репрезентативные примеры, а не полное множество элементарных конъюнкций. Кроме того, возможно, не все элементарные конъюнкции имеют смысл для семантики отношения PAY, и тогда их нужно будет удалить из множества. Наконец, отметим, что это упрощенные варианты элементарных конъюнкций. Согласно определению элементарной конъюнкции, каждый предикат должен входить в нее в естественном виде или в виде отрицания. Значит, m_1, например, следует записать в виде

m_1 : TITLE = "Elect. Eng." ∧ TITLE ≠ "Syst. Anal." ∧ TITLE ≠ "Mech. Eng." ∧ TITLE ≠ "Programmer" ∧ SAL ⩽ 30000

Очевидно, что это избыточно, поэтому мы используем упрощенную форму. Нам также нужна количественная информация о рабочей нагрузке.

1. *Избирательность элементарной конъюнкции*: количество кортежей отношения, удовлетворяющих данной элементарной конъюнкции. Например, избирательность предиката m_2 из примера 2.5 равна 0.25, поскольку ему удовлетворяет один из четырех кортежей PAY. Будем обозначать $sel(m_i)$ избирательность элементарной конъюнкции m_i.
2. *Частота доступа*: частота, с которой пользовательские приложения обращаются к данным. Если $Q = \{q_1, q_2, \ldots, q_q\}$ – множество пользовательских запросов, то $acc(q_i)$ обозначает частоту доступа запроса q_i в течение данного периода.

Заметим, что частоту доступа элементарной конъюнкции можно определить, зная частоты доступа для запросов. Будем обозначать $acc(m_i)$ частоту доступа элементарной конъюнкции.

2.1.1.2. Главная горизонтальная фрагментация

Главная горизонтальная фрагментация применима к отношениям, для которых в графе соединений не существует входящих ребер, и производится с помощью предикатов, определенных на этом отношении. В наших примерах к отношениям PAY и PROJ применима главная горизонтальная фрагментация, а к отношениям EMP и ASG – производная горизонтальная фрагментация. В этом разделе мы займемся главной горизонтальной фрагментацией, а производную отложим до следующего.

Главная горизонтальная фрагментация определяется посредством операции выборки на исходных отношениях схемы базы данных. Если задано отношение R, то его горизонтальные фрагменты определяются как

$$R_i = \sigma_{Fi}(R),\ 1 \leqslant i \leqslant w,$$

где F_i – операторы выборки для получения фрагмента R_i (они еще называются *предикатами фрагментации*). Заметим, что если F_i представлен в конъюнктивной нормальной форме, то это элементарная конъюнкция (m_i). Алгоритм требует, чтобы F_i был элементарной конъюнкцией.

Пример 2.6. Разложение отношения PROJ на горизонтальные фрагменты $PROJ_1$ и $PROJ_2$ в примере 2.1 определяется следующим образом[1]:

$$PROJ_1 = \sigma_{BUDGET \leqslant 200000}(PROJ)$$
$$PROJ_2 = \sigma_{BUDGET > 200000}(PROJ)$$ ♦

В примере 2.6 показана одна из проблем горизонтальной фрагментации. Если области определения атрибутов, участвующих в операторах выборки, непрерывны и бесконечны, то очень трудно определить множество операторов $F = \{F_1, F_2, \ldots, F_n\}$, которое правильно фрагментировало бы отношение. Одно из возможных решений – определить диапазоны, как мы сделали в примере 2.6. Но всегда остается проблема двух крайних точек. Например, если в PROJ нужно вставить новый кортеж, в котором значение BUDGET равно, скажем, $600 000, то придется анализировать фрагментацию, чтобы решить, вставлять ли его в $PROJ_2$ или лучше переопределить старые фрагменты и создать новый следующим образом:

$$PROJ_2 = \sigma_{200000 < BUDGET \wedge BUDGET \leqslant 400000}(PROJ)$$
$$PROJ_3 = \sigma_{BUDGET > 400000}(PROJ)$$

---

[1] Мы предполагаем, что неотрицательность значений BUDGET является свойством отношения, которое гарантируется ограничением целостности. В противном случае следовало бы включить в состав P_r также предикат $0 \leqslant$ BUDGET. Предполагается, что это справедливо для всех примеров и обсуждений в этой главе.

Пример 2.7. Рассмотрим отношение PROJ на рис. 2.2. Мы можем определить следующие горизонтальные фрагменты, основываясь на месте выполнения проекта. Результат показан на рис. 2.6.

$$\text{PROJ}_1 = \sigma_{\text{LOC="Montreal"}}(\text{PROJ})$$
$$\text{PROJ}_2 = \sigma_{\text{LOC="New York"}}(\text{PROJ})$$
$$\text{PROJ}_3 = \sigma_{\text{LOC="Paris"}}(\text{PROJ})$$

♦

PROJ_1

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal

PROJ_2

PNO	PNAME	BUDGET	LOC
P2	Database Develop.	135000	New York
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris

PROJ_3

PNO	PNAME	BUDGET	LOC
P4	Maintenance	310000	Paris

Рис. 2.6 ❖ Главная горизонтальная фрагментация отношения PROJ

Теперь можно определить горизонтальный фрагмент более аккуратно. Горизонтальный фрагмент R_i отношения R состоит из всех кортежей R, удовлетворяющих элементарной конъюнкции m_i. Следовательно, если задано множество элементарных конъюнкций M, то существует столько горизонтальных фрагментов отношения R, сколько элементов в этом множестве. Это множество горизонтальных фрагментов часто называют *множеством минтерм-фрагментов*.

Мы хотим, чтобы множество простых предикатов, образующих элементарную конъюнкцию, было *полным* и *минимальным*. Множество простых предикатов *Pr* называется *полным* тогда и только тогда, когда вероятности доступа со стороны любого приложения к любому кортежу, принадлежащему любому минтерм-фрагменту, определенному в соответствии с *Pr*, равны[1].

Пример 2.8. Рассмотрим фрагментацию отношения PROJ из примера 2.7. Если единственный запрос, обращающийся к PROJ, выбирает кортежи по месту выполнения проекта, то множество полно, т. к. вероятность доступа к любому кортежу любого фрагмента PROJ_i одинакова. Но если имеется второй запрос, который обращается только к кортежам проектов, бюджет которых меньше или равен $200 000, то множество *Pr* неполно. Из-за второго приложения вероятность доступа к некоторым кортежам в каждом фрагменте PROJ_i оказывается выше. Чтобы сделать множество предикатов полным, нужно добавить в *Pr* предикаты (BUDGET ⩽ 200000, BUDGET > 200000):

[1] Очевидно, что определение полноты множества простых предикатов отличается от правила полноты фрагментации, которое мы обсуждали выше.

Pr = {LOC = "Montreal", LOC = "New York", LOC = "Paris",
BUDGET ⩽ 200000, BUDGET > 200000} ♦

Полнота желательна, потому что фрагменты, полученные в соответствии с полным множеством предикатов, логически единообразны, т. к. все они удовлетворяют элементарной конъюнкции. Кроме того, они статистически однородны с точки зрения доступа к ним из приложений. Благодаря этим характеристикам фрагментации нагрузка (при данной смеси приложений) на фрагменты сбалансирована.

Минимальность означает, что если предикат влияет на порядок выполнения фрагментации (т. е. заставляет дополнительно разбить фрагмент *f* на фрагменты f_i и f_j), то должно существовать хотя бы одно приложение, которое обращается к f_i и f_j по-разному. Иными словами, простой предикат должен быть релевантен определению фрагментации. Если все предикаты в множестве *Pr* релевантны, то *Pr* минимально.

Приведем формальное определение релевантности. Пусть m_i и m_j – две элементарные конъюнкции, определения которых различаются только тем, что m_i содержит простой предикат p_i в естественной форме, а m_j – в форме $\neg p_i$. Пусть f_i и f_j – два фрагмента, определенных в соответствии с m_i и m_j. Тогда говорят, что p_i *релевантен*, если

$$\frac{acc(m_i)}{card(f_i)} \neq \frac{acc(m_j)}{card(f_j)}.$$

Пример 2.9. Множество *Pr*, определенное в примере 2.8, полно и минимально. Однако если добавить в *Pr* предикат PNAME = "Instrumentation", то множество перестанет быть минимальным, потому что новый предикат не релевантен относительно *Pr* – не существует двух приложений, которые обращались бы к результирующим фрагментам по-разному. ♦

Теперь мы представим итеративный алгоритм COM_MIN, порождающий полное и минимальное множество предикатов *Pr′*, если задано множество простых предикатов *Pr*. В алгоритме 2.1 используется следующая нотация:

f_i *из Pr′*: фрагмент f_i определяется в соответствии с элементарной конъюнкцией, определенной над предикатами из *Pr′*.

Правило 1: к каждому фрагменту хотя бы одно приложение обращается иначе, чем другие.

Алгоритм COM_MIN начинается с поиска релевантного предиката, который секционирует входное отношение. В цикле **repeat-until** в множество итеративно добавляются предикаты с соблюдением на каждом шаге условия минимальности. Поэтому в конце цикла множество *Pr′* является одновременно минимальным и полным.

На втором шаге процесса проектирования главной горизонтальной фрагментации требуется найти множество элементарных конъюнкций, которые можно определить над предикатами из множества *Pr′*. Эти элементарные конъюнкции определяют фрагменты, которые станут кандидатами на шаге размещения. Определить отдельные элементарные конъюнкции просто,

трудность же в том, что множество таких элементарных конъюнкций может быть очень велико (его размер экспоненциально зависит от числа простых предикатов). Мы увидим, как уменьшить количество элементарных конъюнкций, рассматриваемых при определении фрагментации.

Алгоритм 2.1. COM_MIN

Вход: R: отношение; Pr: множество простых предикатов
Выход: Pr': множество простых предикатов
Объявление: F: множество минтерм-фрагментов
begin
  $Pr' \leftarrow \emptyset$; $F \leftarrow \emptyset$ {инициализация}
  найти $p_i \in Pr$ такое, что p_i фрагментирует R согласно *Правилу* 1
  $Pr' \leftarrow Pr' \cup p_i$
  $Pr \leftarrow Pr - p_i$
  $F \leftarrow F \cup f_i$ {f_i – минтерм-фрагмент в соответствии с pi}
  repeat
    найти $p_j \in Pr$ такие, что p_j фрагментирует некоторые f_k из Pr
    согласно *Правилу* 1
    $Pr' \leftarrow Pr' \cup p_j$
    $Pr \leftarrow Pr - p_j$
    $F \leftarrow F \cup f_j$
    if $\exists p_k \in Pr'$ не являющийся релевантным **then**
      $Pr' \leftarrow Pr' - pk$
      $F \leftarrow F - fk$
    end if
  until *Pr' является полным*
end

Чтобы уменьшить их количество, нужно исключить некоторые элементарные конъюнкции, не имеющие смысла. Для этого мы выявим конъюнкции, противоречащие некоторому множеству импликаций I. Например, если $Pr' = \{p_1, p_2\}$, где

$p_1 : att = value\_1$
$p_2 : att = value\_2$

и область определения атрибута att равна {$value\_1$, $value\_2$}, то I содержит две импликации:

$i_1 : (att = value\_1) \Rightarrow \neg(att = value\_2)$
$i_2 : \neg(att = value\_1) \Rightarrow (att = value\_2)$

В соответствии с Pr' определены следующие четыре элементарные конъюнкции:

$m_1 : (att = value\_1) \wedge (att = value\_2)$
$m_2 : (att = value\_1) \wedge \neg(att = value\_2)$
$m_3 : \neg(att = value\_1) \wedge (att = value\_2)$
$m_4 : \neg(att = value\_1) \wedge \neg(att = value\_2)$

В этом случае элементарные конъюнкции m_1 и m_4 противоречат импликациям I и потому должны быть исключены из M.

Алгоритм 2.2 представляет собой алгоритм главной горизонтальной фрагментации, называемый PHORIZONTAL. Входом является отношение R, подлежащее главной горизонтальной фрагментации, и множество простых предикатов Pr, найденное в соответствии с приложениями, обращающимися к R.

Алгоритм 2.2. PHORIZONTAL

Вход: R: отношение; Pr: множество простых предикатов
Выход: F_R: множество горизонтальных фрагментов R
begin
  $Pr' \leftarrow$COM_MIN(R, Pr)
  определить множество M элементарных конъюнкций
  определить множество I импликаций среди $pi \in Pr'$
  foreach $m_i \in M$ **do**
    if m_i *противоречиво относительно* I **then**
      $M \leftarrow M - mi$
    end if
  end foreach
  $F_R = \{R_i | R_i = \sigma_{mi} R\}, \forall m_i \in M$
end

Пример 2.10. Рассмотрим отношения PAY и PROJ на рис. 2.5, подлежащие главной горизонтальной фрагментации.

Предположим, что к PAY обращается только один запрос, который проверяет информацию о запросе и определяет, на сколько ее повысить. Допустим, что записи о работниках находятся в двух местах: в одном те, у которых зарплата меньше или равна $30 000, а в другом – с зарплатами больше $30 000. Таким образом, запрос будет выполняться в двух узлах.

Простые предикаты, необходимые для секционирования отношения PAY, имеют вид:

p_1 : SAL ⩽ 30000
p_2 : SAL > 30000

так что начальное множество простых предикатов $Pr = \{p_1, p_2\}$. Применение алгоритма COM_MIN с $i = 1$ в качестве начального значения дает $Pr = \{p_1\}$. Это множество полное и минимальное, потому что p_2 не фрагментировал бы f_1 (минтерм-фрагмент, образованный относительно p_1) согласно правилу 1. Мы можем образовать следующие элементарные конъюнкции, являющиеся элементами M:

m_1 : SAL < 30000
m_2 : ¬(SAL ⩽ 30000) = SAL > 30000

Поэтому в соответствии с M мы определяем два фрагмента $F_{PAY} = \{PAY_1, PAY_2\}$ (рис. 2.7).

PAY_1

TITLE	SAL
Mech. Eng.	27000
Programmer	24000

PAY_2

TITLE	SAL
Elect. Eng.	40000
Syst. Anal.	34000

Рис. 2.7 ❖ Горизонтальная фрагментация отношения PAY

Далее рассмотрим отношение PROJ. Предположим, что имеется два запроса. Первый предъявляется в двух узлах и находит названия и бюджеты проектов по месту выполнения. На языке SQL этот запрос имеет вид

```
SELECT PNAME, BUDGET
FROM   PROJ
WHERE  LOC=Value
```

Относительно этого приложения простые предикаты таковы:

p_1 : LOC = "Montreal"
p_2 : LOC = "New York"
p_3 : LOC = "Paris"

Второй запрос предъявляется в двух узлах и касается управления проектами. Проекты с бюджетом, меньшим или равным $200 000, хранятся в первом узле, а с большим бюджетом – во втором. Следовательно, относительно второго приложения простые предикаты, применяемые для фрагментации, имеют вид:

p_4 : BUDGET $\leqslant$ 200000
p_5 : BUDGET > 200000

Применив алгоритм COM_MIN, получаем полное и минимальное множество $Pr' = \{p_1, p_2, p_4\}$. На самом деле COM_MIN мог бы добавить в Pr' любые два из предикатов p_1, p_2, p_3; в этом примере мы решили включить p_1 и p_2.

Зная Pr', можно определить следующие шесть элементарных конъюнкций, составляющих M:

m_1 : (LOC = "Montreal") $\wedge$ (BUDGET $\leqslant$ 200000)
m_2 : (LOC = "Montreal") $\wedge$ (BUDGET > 200000)
m_3 : (LOC = "New York") $\wedge$ (BUDGET $\leqslant$ 200000)
m_4 : (LOC = "New York") $\wedge$ (BUDGET > 200000)
m_5 : (LOC = "Paris") $\wedge$ (BUDGET $\leqslant$ 200000)
m_6 : (LOC = "Paris") $\wedge$ (BUDGET > 200000)

Как было отмечено в примере 2.5, это не единственный возможный способ сгенерировать элементарные конъюнкции. Например, можно было бы определить предикаты вида

$$p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5.$$

Однако очевидные импликации (например, $p_1 \Rightarrow \neg p_2 \wedge \neg p_3$, $\neg p_5 \Rightarrow p_4$) исключают эти предикаты, оставляя только $m_1, \ldots, m_6$.

Глядя на базу данных на рис. 2.2, возникает искушение заявить, что истинны следующие импликации:

i_8 : LOC = "Montreal" ⇒ ¬(BUDGET > 200000)
i_9 : LOC = "Paris" ⇒ ¬(BUDGET ⩽ 200000)
i_{10} : ¬(LOC = "Montreal") ⇒ BUDGET ⩽ 200000
i_{11} : ¬(LOC = "Paris") ⇒ BUDGET > 200000

Вспомним, однако, что импликации следует определять согласно семантике базы данных, а не согласно текущим значениям. В семантике базы данных нет ничего такого, из чего следовала бы истинность импликаций i_8–i_{11}. Некоторые фрагменты, определенные в соответствии с множеством $M = \{m_1, \ldots, m_6\}$, могут быть пусты, но все равно остаются фрагментами.

В результате главной горизонтальной фрагментации PROJ образовано шесть фрагментов $F_{PROJ} = \{PROJ_1, PROJ_2, PROJ_3, PROJ_4, PROJ_5, PROJ_6\}$ отношения PROJ в соответствии с элементарными конъюнкциями M (рис. 2.8). Поскольку фрагменты $PROJ_2$ и $PROJ_5$ пусты, на рис. 2.8 они не показаны. ♦

$PROJ_1$

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal

$PROJ_2$

PNO	PNAME	BUDGET	LOC
P2	Database Develop.	135000	New York

$PROJ_3$

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	250000	New York

$PROJ_4$

PNO	PNAME	BUDGET	LOC
P4	Maintenance	310000	Paris

Рис. 2.8 ❖ Горизонтальная фрагментация отношения PROJ

2.1.1.3. Производная горизонтальная фрагментация

Производная горизонтальная фрагментация применяется к целевым отношениям в графе соединений и производится на основе предикатов, определенных над исходным отношением ребра графа соединений. В наших примерах отношения EMP и ASG подлежат производной горизонтальной фрагментации. Напомним, что ребро между исходным и целевым отношениями определяется как эквисоединение, которое можно реализовать с помощью полусоединений. Этот момент важен, поскольку мы хотим фрагментировать целевое отношение в соответствии с фрагментацией его исходного, но при этом желательно, чтобы результирующий фрагмент был определен *только* по атрибутам целевого отношения.

Поэтому если задано ребро L, где $source(L) = \mathtt{S}$ и $target(L) = \mathtt{R}$, то производные горизонтальные фрагменты R определяются следующим образом:

$$\mathtt{R}_i = \mathtt{R} \ltimes \mathtt{S}_i,\ 1 \leqslant i \leqslant w,$$

где w – максимальное число фрагментов, определенных на R, а $\mathtt{S}_i = \sigma_{Fi}(\mathtt{S})$, где F_i – формула, по которой определен главный горизонтальный фрагмент $\mathtt{S}_i$.

Пример 2.11. Рассмотрим ребро L_1 на рис. 2.5, для которого $source(L_1)$ = PAY и $target(L_1)$ = EMP. Тогда инженеров можно объединить в две группы согласно зарплате: те, чья зарплата меньше или равна \$30 000, и те, кто зарабатывает больше \$30 000. Фрагменты $\mathtt{EMP}_1$ и $\mathtt{EMP}_2$ определены следующим образом:

$$\mathtt{EMP}_1 = \mathtt{EMP} \ltimes \mathtt{PAY}_1$$
$$\mathtt{EMP}_2 = \mathtt{EMP} \ltimes \mathtt{PAY}_2$$

где

$$\mathtt{PAY}_1 = \sigma_{\mathtt{SAL} \leqslant 30000}(\mathtt{PAY})$$
$$\mathtt{PAY}_2 = \sigma_{\mathtt{SAL} > 30000}(\mathtt{PAY})$$

Результат такой фрагментации показан на рис. 2.9. ♦

$\mathtt{EMP}_1$

ENO	ENAME	TITLE
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E7	R. Davis	Mech. Eng.

$\mathtt{EMP}_2$

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E2	M. Smith	Syst. Anal.
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E8	J. Jones	Syst. Anal.

Рис. 2.9 ❖ Производная горизонтальная фрагментация отношения EMP

Для выполнения производной горизонтальной фрагментации нужно задать три вида входных данных: множество фрагментов исходного отношения (например, $\mathtt{PAY}_1$ и $\mathtt{PAY}_2$ в примере 2.11), целевое отношение и множество предикатов полусоединения между исходным и целевым отношениями (например, EMP.TITLE = PAY.TITLE в примере 2.11). Сам алгоритм фрагментации тривиален, поэтому мы не будем детально описывать его.

Существует одно потенциальное осложнение, заслуживающее внимания. Часто бывает, что в схеме базы данных имеется несколько ребер, входящих в отношение R (например, на рис. 2.5 в отношение ASG входят два ребра). В таком случае имеется несколько возможных производных фрагментаций R. Выбор одной из них основан на двух критериях:

1) фрагментация с лучшими характеристиками соединения;
2) фрагментация, используемая в большем числе запросов.

Сначала обсудим второй критерий. Он вполне очевиден, если принять во внимание частоту обращения к данным при имеющейся рабочей нагрузке. По возможности следует упростить доступ «тяжелым» пользователям, чтобы свести к минимуму их влияние на общую производительность системы.

Но применение первого критерия не столь очевидно. Рассмотрим, к примеру, фрагментацию, которую мы обсуждали в примере 2.1. Результат (и цель) этой фрагментации состоит в том, чтобы ускорить соединение отношений EMP и PAY двумя способами: (1) выполняя его над меньшими отношениями (т. е. фрагментами) и (2) по возможности распараллелив операцию соединения.

С первым пунктом все понятно. Что же касается второго, то мы имеем дело с внутризапросным параллелизмом соединения, т. е. выполнением всех соединений параллельно, что возможно лишь при определенных условиях. Рассмотрим, к примеру, ребра между фрагментами EMP и PAY из примера 2.9. Мы имеем $PAY_1 \rightarrow EMP_1$ и $PAY_2 \rightarrow EMP_2$, т. е. в каждый фрагмент входит и из каждого фрагмента выходит только одно ребро, так что это *простой* граф соединений. Преимущество структуры, в которой соединение между фрагментами простое, в том, что источник и цель ребра можно разместить в одном узле, а соединения между разными парами фрагментов можно обрабатывать независимо и параллельно.

К сожалению, свести дело к простым графам соединений не всегда возможно. Тогда следующая в порядке предпочтительности альтернатива – найти структуру, которая приводит к *разрезанному* графу соединений. Разрезанный граф состоит из двух или более подграфов, между которыми нет ни одного ребра. Полученные таким образом фрагменты нельзя распределить для параллельного выполнения так просто, как в случае простых графов соединений, но размещение все же возможно.

Пример 2.12. Продолжим проектировать распределение базы данных, начатое в примере 2.10. Мы уже решили, как фрагментировать отношение EMP в соответствии с фрагментацией PAY (пример 2.11). Теперь рассмотрим отношение ASG. Предположим, что имеется два запроса.

1. Первый запрос находит имена инженеров, работающих в определенных местах. Он выполняется во всех трех узлах и обращается к информации об инженерах, занятых в локальных проектах с большей вероятностью, чем в проектах, выполняемых в других местах.
2. В каждом административном узле, где хранятся записи о работниках, пользователи хотели бы знать, какие функции работник выполняет в проекте и сколько времени над ним работает.

Первый запрос приводит к фрагментации ASG в соответствии с (непустыми) фрагментами $PROJ_1$, $PROJ_3$, $PROJ_4$ и $PROJ_6$ отношения PROJ, полученными в примере 2.10:

$$PROJ_1 : \sigma_{LOC="Montreal" \wedge BUDGET \leq 200000}(PROJ)$$
$$PROJ_3 : \sigma_{LOC="New\ York" \wedge BUDGET} \leqslant 200000(PROJ)$$
$$PROJ_4 : \sigma_{LOC="New\ York" \wedge BUDGET > 200000}(PROJ)$$
$$PROJ_6 : \sigma_{LOC="Paris" \wedge BUDGET > 200000}(PROJ)$$

Поэтому производная фрагментация ASG, согласованная с $\{PROJ_1, PROJ_3, PROJ_4, PROJ_6\}$, определена следующим образом:

$$ASG_1 = ASG \ltimes PROJ_1$$
$$ASG_2 = ASG \ltimes PROJ_3$$

$ASG_3 = ASG \ltimes PROJ_4$
$ASG_4 = ASG \ltimes PROJ_6$

Эти фрагменты показаны на рис. 2.10.

ASG_1

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24

ASG_2

ENO	PNO	RESP	DUR
E2	P2	Analyst	6
E4	P2	Programmer	18
E5	P2	Manager	24

ASG_3

ENO	PNO	RESP	DUR
E3	P3	Consultant	10
E7	P3	Engineer	36
E8	P3	Manager	40

ASG_4

ENO	PNO	RESP	DUR
E3	P4	Engineer	48
E6	P4	Manager	48

Рис. 2.10 ❖ Производная фрагментация ASG относительно PROJ

Второй запрос на SQL формулируется так:

```
SELECT RESP, DUR
FROM   ASG NATURAL JOIN EMP_i
```

где $i = 1$ или $i = 2$ в зависимости от того, в каком узле выполняется запрос. Производная фрагментация ASG, согласованная с фрагментацией EMP, определена ниже и показана на рис. 2.11.

$ASG_1 = ASG \ltimes EMP_1$
$ASG_2 = ASG \ltimes EMP_2$

ASG_1

ENO	PNO	RESP	DUR
E3	P3	Consultant	10
E3	P4	Engineer	48
E4	P2	Programmer	18
E7	P3	Engineer	36

ASG_2

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E5	P2	Manager	24
E6	P4	Manager	48
E8	P3	Manager	40

Рис. 2.11 ❖ Производная фрагментация ASG относительно EMP

Этот пример позволяет сделать два наблюдения:

1) производная фрагментация может производиться по цепочке, в которой одно отношение фрагментируется как результат дизайна другого и, в свою очередь, определяет фрагментацию следующего отношения (как в цепочке PAY → EMP → ASG);
2) обычно бывает более одной потенциальной фрагментации отношения (как в случае ASG). Окончательный выбор схемы фрагментации можно сделать на этапе размещения.

2.1.1.4. Проверка корректности

Теперь проверим корректность рассмотренных выше алгоритмов фрагментации относительно сформулированных критериев.

Полнота

Полнота главной горизонтальной фрагментации основана на используемых предикатах выборки. Коль скоро предикаты выборки полны, гарантируется, что и результирующая фрагментации будет полна. Поскольку в основе алгоритма фрагментации лежит множество *полных* и *минимальных* предикатов (Pr'), то полнота гарантирована, если Pr' определено надлежащим образом.

Полноту производной горизонтальной фрагментации установить несколько труднее, потому что предикат, определяющий фрагментацию, включает два отношения.

Пусть R – целевое отношение ребра, в котором исходным является отношение S, причем R и S фрагментированы в виде $F_R = \{R_1, R_2, \ldots, R_w\}$ и $F_S = \{S_1, S_2, \ldots, S_w\}$ соответственно. Пусть A – атрибут, по которому соединяются R и S. Тогда для любого кортежа t отношения R_i должен существовать кортеж t' отношения S_i такой, что $t[A] = t'[A]$. Это хорошо известное правило *ссылочной целостности*, гарантирующее, что кортежи любого фрагмента целевого отношения имеются также и в исходном отношении. Например, в ASG не должно существовать кортежа с номером проекта, отсутствующим в PROJ. Аналогично в EMP не должно существовать кортежей, в которых значение TITLE не встречается в PAY.

Реконструкция

Реконструкция глобального отношения из его фрагментов осуществляется оператором объединения как в главной, так и в производной горизонтальной фрагментации. Таким образом, для отношения R с фрагментацией $F_R = \{R_1, R_2, \ldots, R_w\}$, $R = \cup R_i$, $\forall R_i \in F_R$.

Дизъюнктность

Доказать дизъюнктность для главной горизонтальной фрагментации проще, чем для производной. В первом случае дизъюнктность гарантируется при условии, что элементарные конъюнкции, определяющие фрагментацию, взаимно исключающие.

Но в производной фрагментации имеется полусоединение, и оно вносит немалую сложность. Дизъюнктность можно гарантировать, если граф соединений простой. В противном случае нужно исследовать фактические значения кортежей. В общем случае мы не хотим, чтобы кортеж целевого отношения соединялся с двумя или более кортежами исходного отношения, если эти кортежи находятся в разных фрагментах последнего. Установить это не всегда легко, и именно поэтому всегда предпочтительны схемы производной фрагментации, порождающие простой граф соединений.

Пример 2.13. При фрагментации отношения PAY (пример 2.10) были такие элементарные конъюнкции $M = \{m_1, m_2\}$:

m_1 : SAL ⩽ 30000
m_2 : SAL > 30000

Поскольку m_1 и m_2 взаимно исключающие, эта фрагментация PAY дизъюнктная.

Но для отношения EMP мы требуем, чтобы:

1) у каждого инженера была только одна должность;
2) с каждой должностью была связана только одна зарплата.

Поскольку оба этих правила вытекают из семантики базы данных, фрагментация EMP относительно PAY также дизъюнктная. ♦

2.1.2. Вертикальная фрагментация

Напомним, что вертикальная фрагментация отношения R порождает фрагменты R_1, R_2, ..., R_r, каждый из которых содержит подмножество атрибутов R, а также первичный ключ R. Как и в случае горизонтальной фрагментации, цель состоит в том, чтобы разделить отношение на множество меньших отношений, так чтобы как можно больше пользовательских приложений работало только с одним фрагментом. Первичный ключ включается в каждый фрагмент, чтобы сделать возможной реконструкцию, о чем мы поговорим позже. Это также полезно для обеспечения целостности, поскольку первичный ключ функционально определяет все атрибуты отношения; наличие его в каждом фрагменте позволяет обойтись без распределенных вычислений для поддержания ограничения первичного ключа.

Вертикальное секционирование принципиально сложнее горизонтального в основном из-за большего числа вариантов. Например, при горизонтальном секционировании, если общее число простых предикатов в множестве *Pr* равно n, существует 2^n элементарных конъюнкций. Но при вертикальном секционировании, если отношение содержит m атрибутов, не входящих в состав первичного ключа, число возможных фрагментов равно $B(m)$ – m-му числу Белла. При больших m $B(m) \approx m^m$; например, для $m = 10$ $B(m) \approx 115\,000$, для $m = 15$ $B(m) \approx 10^9$, а для $m = 30$ $B(m) \approx 10^{23}$.

Отсюда следует, что любая попытка найти оптимальное решение задачи вертикального секционирования обречена на провал, поэтому приходится прибегать к эвристическим методам. Есть два эвристических подхода к вертикальной фрагментации глобальных отношений[1]:

1) *группировка* – сначала каждому атрибуту сопоставляется фрагмент, а затем на каждом шаге некоторые фрагменты объединяются, пока не будет выполнено заданное условие;
2) *расщепление* – в начале берется само отношение и принимается решение, как его лучше разбить, исходя из способов доступа к атрибутам со стороны приложений.

---

[1] Существует и третий, экстремальный подход, применяемый в столбцовых СУБД (типа MonetDB и Vertica), когда каждый столбец отображается на один фрагмент. Но поскольку в этой книге столбцовые СУБД не рассматриваются, то обсуждать этот подход мы не будем.

Далее мы будем обсуждать только технику расщепления, поскольку она более естественно укладывается в принятую нами методику проектирования, ибо «оптимальное» решение, вероятно, ближе к полному отношению, чем к множеству фрагментов, состоящих из одного атрибута. Кроме того, расщепление порождает неперекрывающиеся фрагменты, тогда как группировка часто приводит к перекрывающимся. Из-за требования дизъюнктности мы предпочитаем неперекрывающиеся фрагменты. Разумеется, говоря о том, что фрагменты не перекрываются, мы не учитываем атрибуты, составляющие первичный ключ.

2.1.2.1. Требования к дополнительной информации

Как и раньше, нам нужна информация о рабочей нагрузке. Поскольку при вертикальном секционировании в один фрагмент помещаются атрибуты, доступ к которым обычно производится совместно, то нужна какая-то мера «совместности». Этой мерой является *родство* (affinity) атрибутов, показывающее, насколько тесно они связаны между собой. Трудно ожидать, что проектировщик или пользователи смогут точно указать эту величину. Мы покажем, как ее можно получить из более примитивных данных.

Обозначим $Q = \{q_1, q_2, \ldots, q_q\}$ множество пользовательских запросов, обращающихся к отношению $R(A_1, A_2, \ldots, A_n)$. Тогда с каждым запросом q_i и каждым атрибутом A_j ассоциируется *показатель использования атрибута* $use(q_i, A_j)$:

$$use(q_i, A_j) = \begin{cases} 1, \text{ если атрибут } A_j \text{ упоминается в запросе } q_i, \\ 0 \text{ в противном случае.} \end{cases}$$

Вычислить векторы $use(q_i, \bullet)$ для каждого запроса нетрудно.

Пример 2.14. Рассмотрим отношение PROJ на рис. 2.2. Предположим, что к нему предъявляются перечисленные ниже запросы. Для каждого запроса приводится его выражение на языке SQL.

q_1: найти бюджет проекта, зная его идентификационный номер.

```
SELECT BUDGET
FROM   PROJ
WHERE  PNO=Value
```

q_2: найти названия и бюджеты всех проектов.

```
SELECT PNAME, BUDGET
FROM   PROJ
```

q_3: найти названия проектов, выполняемых в данном городе.

```
SELECT PNAME
FROM   PROJ
WHERE  LOC=Value
```

q_4: найти суммарный бюджет проектов, выполняемых в данном городе.

```
SELECT SUM(BUDGET)
FROM   PROJ
WHERE  LOC=Value
```

Показатели использования атрибутов для этих четырех запросов можно представить в виде матрицы (рис. 2.12), элемент (i, j) которой равен $use(q_i, \mathtt{A}_j)$. ♦

	PNO	PNAME	BUDGET	LOC
q_1	0	1	1	0
q_2	1	1	1	0
q_3	1	0	0	1
q_4	0	0	1	0

Рис. 2.12 ❖ Пример матрицы использования атрибутов

Показатели использования атрибутов недостаточно общие для того, чтобы лечь в основу расщепления и фрагментации, поскольку не отражают частоту выполнения приложения. Эту частоту можно включить в определение меры родства $aff(\mathtt{A}_i, \mathtt{A}_j)$, которая измеряет связь между атрибутами отношения в соответствии с тем, как к ним обращаются запросы.

Родство атрибутов $\mathtt{A}_i$ и $\mathtt{A}_j$ отношения $\mathtt{R}(\mathtt{A}_1, \mathtt{A}_2, \ldots, \mathtt{A}_n)$ относительно множества запросов $Q = \{q_1, q_2, \ldots, q_q\}$ определяется следующим образом:

$$aff(\mathtt{A}_i, \mathtt{A}_j) = \sum_{k | use(q_k, \mathtt{A}_i)=1 \wedge use(q_k, \mathtt{A}_j)=1} \sum_{\forall S_l} ref_l(q_k) acc_l(q_k),$$

где $ref_l(q_k)$ – количество обращений к атрибутам ($\mathtt{A}_i$, $\mathtt{A}_j$) для каждого выполнения приложения q_k в узле S_l, а $acc_l(q_k)$ – частота доступа со стороны приложений, которая была определена ранее и модифицирована с учетом частот в разных узлах.

Результатом этого вычисления является матрица $n \times n$, элементами которой являются определенные выше числа. Она называется *матрицей родства атрибутов* (*AA*).

Пример 2.15. Продолжим рассмотрение, начатое в примере 2.14. Для простоты предположим, что $ref_l(q_k) = 1$ для всех q_k и S_l. Если частоты приложений равны

$$\begin{array}{ll} acc_1(q_1) = 15 & acc_1(q_2) = 5 \\ acc_1(q_3) = 25 & acc_1(q_4) = 3 \\ acc_2(q_1) = 20 & acc_2(q_2) = 0 \\ acc_2(q_3) = 25 & acc_3(q_4) = 0 \\ acc_3(q_1) = 10 & acc_3(q_2) = 0 \\ acc_3(q_3) = 25 & acc_2(q_4) = 0 \end{array}$$

то меру родства атрибутов PNO и BUDGET можно выразить в виде

$$aff(\mathtt{PNO}, \mathtt{BUDGET}) = \sum_{k=1}^{1} \sum_{l=1}^{3} acc_l(q_k) = acc_1(q_1) + acc_2(q_1) + acc_3(q_1) = 45,$$

поскольку единственным приложением, которое обращается к обоим атрибутам, является q_1. Полностью матрица родства атрибутов показана на рис. 2.13. Отметим, что значения на диагонали не вычисляются, поскольку не имеют смысла.

	PNO	PNAME	BUDGET	LOC
PNO	–	0	45	0
PNAME	0	–	5	75
BUDGET	45	5	–	3
LOC	0	75	3	–

Рис. 2.13 ❖ Пример матрицы родства атрибутов

До конца этой главы матрица родства атрибутов будет использоваться как ориентир, направляющий наши усилия по фрагментации. Идея в том, чтобы сначала собрать вместе атрибуты с высокой степенью родства, а затем соответственно расщепить отношение.

2.1.2.2. Алгоритм кластеризации

Главная задача при проектировании алгоритма вертикальной фрагментации – найти способ группировки атрибутов отношения на основе элементов матрицы родства атрибутов. Мы обсудим предложенный для этой цели алгоритм энергии связей (bond energy algorithm – BEA). Но можно использовать и другие алгоритмы кластеризации.

BEA принимает матрицу родства атрибутов для отношения $R(A_1, \ldots, A_n)$, переставляет ее строки и столбцы и порождает *кластерную матрицу родства* (*CA*). Перестановки производятся таким образом, чтобы *максимизировать* следующую *меру глобального родства* (*AM*):

$$AM = \sum_{i=1}^{n}\sum_{j=1}^{n} aff(A_i, A_j)[aff(A_i, A_{j-1}) + aff(A_i, A_{j+1}) + aff(A_{i-1}, A_j) + aff(A_{i+1}, A_j)],$$

где

$$aff(A_0, A_j) = aff(A_i, A_0) = aff(A_{n+1}, A_j) = aff(A_i, A_{n+1}) = 0.$$

Последний набор условий учитывает случаи, когда атрибут помещается в *CA* слева от самого левого атрибута или справа от самого правого при перестановке столбцов, либо перед первой строкой или после последней строки при перестановке строк. Будем обозначать A_0 атрибут слева от самого левого атрибута и строку перед первой строкой, а A_{n+1} – атрибут справа от самого правого атрибута или строку после последней строки. В этих случаях мы полагаем равными 0 значения *aff* между атрибутом, рассматриваемым для помещения, и его левым и правым (верхним или нижним) соседями, поскольку их не существует в *CA*.

Функция максимизации учитывает только ближайших соседей, поэтому большие значения группируются с большими, а малые с малыми. Кроме

того, матрица родства атрибутов (*AA*) симметрична, так что целевая функция сводится к

$$AM = \sum_{i=1}^{n}\sum_{j=1}^{n} aff(\mathsf{A}_i,\mathsf{A}_j)[aff(\mathsf{A}_i,\mathsf{A}_{j-1}) + aff(\mathsf{A}_i,\mathsf{A}_{j+1})].$$

Детали алгоритма BEA приведены в алгоритме 2.3. Порождение кластерной матрицы родства (*CA*) состоит из трех шагов.

1. *Инициализация*. Поместить и зафиксировать произвольно выбранный столбец *AA* в *CA*. Мы взяли столбец 1.
2. *Итерация*. Выбрать один из оставшихся $n - i$ столбцов (где i – количество столбцов, уже помещенных в *CA*) и попытаться поместить их в оставшиеся $i + 1$ позиций матрицы *CA*. Выбрать такое место для размещения, при котором вносится максимальный вклад в описанную выше меру глобального родства. Повторять этот шаг, пока не останется столбцов, которые можно было бы поместить.
3. *Упорядочение строк*. После того как порядок столбцов определен, следует изменить порядок строк, так чтобы их относительные позиции совпадали с относительными позициями столбцов[1].

Алгоритм 2.3. BEA

```
Вход: AA: матрица родства атрибутов
Выход: CA: кластерная матрица родства
begin
    {инициализация; запомнить, что AA – матрица n×n}
    CA(•, 1) ← AA(•, 1)
    CA(•, 2) ← AA(•, 2)
    index ← 3
    while index ⩽ n do                    {выбрать «лучшее» место для атрибута AA_index}
        for i from 1 to index − 1 by 1 do вычислить cont(A_{i−1}, A_index, A_i)
        вычислить cont(A_{index−1}, A_index, A_{index+1})              {граничное условие}
        loc ← размещение, определяемое максимальным значением cont
        for j from index to loc by −1 do
            CA(•, j) ← CA(•, j − 1)                    {перетасовать обе матрицы}
        end for
        CA(•, loc) ← AA(•, index)
        index ← index + 1
    end while
    упорядочить строки в соответствии с относительным порядком столбцов
end
```

[1] Начиная с этого момента мы можем обозначать элементы матриц *AA* и *CA* как $AA(i, j)$ и $CA(i, j)$ соответственно. Отображение на меры родства имеет вид $AA(i, j) = aff(\mathsf{A}_i, \mathsf{A}_j)$ и $CA(i, j)$ = *aff*(атрибут, помещенный в столбец i матрицы *CA*, атрибут, помещенный в столбец j матрицы *CA*). Матрицы *AA* и *CA* совпадают во всем, кроме порядка атрибутов, но поскольку алгоритм упорядочивает столбцы *CA* раньше, чем строки, то мера родства *CA* задается относительно столбцов. Заметим, что условие в концевых точках для вычисления меры родства (*AM*) можно в этой нотации записать в виде $CA(0, j) = CA(i, 0) = CA(n + 1, j) = CA(i, n + 1) = 0$.

Чтобы второй шаг алгоритма работал, нужно определить, что понимается под вкладом атрибута в меру родства. Этот вклад можно вычислить следующим образом. Напомним, что мера глобального родства *AM* выше была определена как

$$AM = \sum_{i=1}^{n}\sum_{j=1}^{n} aff(\mathtt{A}_i,\mathtt{A}_j)[aff(\mathtt{A}_i,\mathtt{A}_{j-1}) + aff(\mathtt{A}_i,\mathtt{A}_{j+1})],$$

что можно переписать в виде

$$\begin{aligned} AM &= \sum_{i=1}^{n}\sum_{j=1}^{n}[aff(\mathtt{A}_i,\mathtt{A}_j)aff(\mathtt{A}_i,\mathtt{A}_{j-1}) + aff(\mathtt{A}_i,\mathtt{A}_j)aff(\mathtt{A}_i,\mathtt{A}_{j+1})] \\ &= \sum_{j=1}^{n}\left[\sum_{i=1}^{n} aff(\mathtt{A}_i,\mathtt{A}_j)aff(\mathtt{A}_i,\mathtt{A}_{j-1}) + \sum_{i=1}^{n}[aff(\mathtt{A}_i,\mathtt{A}_j)aff(\mathtt{A}_i,\mathtt{A}_{j+1})]\right]. \end{aligned}$$

Определим *связь* между двумя атрибутами $\mathtt{A}_x$ и $\mathtt{A}_y$:

$$bond(\mathtt{A}_x,\mathtt{A}_y) = \sum_{z=1}^{n} aff(\mathtt{A}_z,\mathtt{A}_x)aff(\mathtt{A}_z,\mathtt{A}_y).$$

Тогда *AM* можно записать в виде

$$AM = \sum_{j=1}^{n}[bond(\mathtt{A}_j,\mathtt{A}_{j-1}) + bond(\mathtt{A}_j,\mathtt{A}_{j+1})].$$

Теперь рассмотрим следующие *n* атрибутов:

$$\underbrace{\mathtt{A}_1\mathtt{A}_2 \ldots \mathtt{A}_{i-1}}_{AM'}\,\mathtt{A}_i\mathtt{A}_j\,\underbrace{\mathtt{A}_{j+1} \ldots \mathtt{A}_n}_{AM^1}.$$

Меру глобального родства для этих атрибутов можно записать так:

$$\begin{aligned} AM_{old} &= AM' + AM^1 \\ &\quad + bond(\mathtt{A}_{i-1},\mathtt{A}_i) + bond(\mathtt{A}_j,\mathtt{A}_i) + bond(\mathtt{A}_j,\mathtt{A}_{j+1}) \\ &= \sum_{l=1}^{i}[bond(\mathtt{A}_l,\mathtt{A}_{l-1}) + bond(\mathtt{A}_l,\mathtt{A}_{l+1})] \\ &\quad + \sum_{l=i+2}^{n}[bond(\mathtt{A}_l,\mathtt{A}_{l-1}) + bond(\mathtt{A}_l,\mathtt{A}_{l+1})] \\ &\quad + 2bond(\mathtt{A}_i,\mathtt{A}_j). \end{aligned}$$

Рассмотрим помещение нового атрибута $\mathtt{A}_k$ в кластерную матрицу родства между атрибутами $\mathtt{A}_i$ и $\mathtt{A}_j$. Новую меру глобального родства можно точно так же записать в виде:

$$\begin{aligned} AM_{new} &= AM' + AM^1 + bond(\mathtt{A}_i,\mathtt{A}_k) + bond(\mathtt{A}_k,\mathtt{A}_i) \\ &\quad + bond(\mathtt{A}_k,\mathtt{A}_j) + bond(\mathtt{A}_j,\mathtt{A}_k) \\ AM_{new} &= AM' + AM^1 + 2bond(\mathtt{A}_i,\mathtt{A}_k) + 2bond(\mathtt{A}_k,\mathtt{A}_j). \end{aligned}$$

Таким образом, вклад в меру глобального родства, получающийся в результате помещения атрибута $\mathtt{A}_k$ между $\mathtt{A}_i$ и $\mathtt{A}_j$, равен

$$cont(A_i, A_k, A_j) = AM_{new} - AM_{old}$$
$$= 2bond(A_i, A_k) + 2bond(A_k, A_j) - 2bond(A_i, A_j).$$

Пример 2.16. Рассмотрим матрицу *AA* на рис. 2.13 и изучим вклад, который дает перемещение атрибута LOC в позицию между атрибутами PNO и PNAME, он описывается формулой

cont (PNO, LOC, PNAME) = 2*bond*(PNO, LOC) + 2*bond*(LOC, PNAME) – 2*bond*(PNO, PNAME).

Вычисляем все члены

bond(PNO, LOC) = 45 ∗ 0 + 0 ∗ 75 + 45 ∗ 3 + 0 ∗ 78 = 135;
bond(LOC, PNAME) = 11 865;
bond(PNO, PNAME) = 225.

Следовательно,

cont (PNO, LOC, PNAME) = 2 ∗ 135 + 2 ∗ 11 865 – 2 ∗ 225 = 23 550. ♦

До сих пор при обсуждении алгоритма в центре нашего внимания были столбцы матрицы родства атрибутов. Но можно переработать алгоритм так, чтобы он работал со строками. Поскольку матрица *AA* симметрична, оба варианта дают одинаковый результат.

Заметим, что алгоритм 2.3 на шаге инициализации помещает второй столбец рядом с первым. Понятно, что это работает, потому что связь между обоими не зависит от их относительного расположения.

Вычисление *cont* в концевых точках требует аккуратности. Если рассматривается помещение атрибута A_i слева от самого левого атрибута, то приходится вычислять связь между несуществующим левым элементом и A_k [т. е. *bond*(A_0, A_k)]. Таким образом, мы должны воспользоваться условиями, сформулированными в определении меры глобального родства *AM*, где *CA*(0, *k*) = 0. Аналогичные рассуждения относятся к помещению справа от самого правого атрибута.

Пример 2.17. Рассмотрим кластеризацию атрибутов отношения PROJ и использование матрицы родства атрибутов *AA* на рис. 2.13.

На шаге инициализации мы копируем столбцы 1 и 2 матрицы *AA* в матрицу *CA* (рис. 2.14a) и начинаем со столбца 3 (т. е. атрибута BUDGET). Столбец 3 можно поместить в одно из трех мест: слева от столбца 1, получив порядок (3-1-2), между столбцами 1 и 2, что дает порядок (1-3-2), и справа от столбца 2, получив порядок (1-2-3). Отметим, что при вычислении вклада последнего порядка необходимо вычислять *cont*(PNAME, BUDGET, LOC), а не *cont*(PNO, PNAME, BUDGET). Однако атрибут LOC еще не помещен в матрицу *CA* (рис. 2.14b), что заставляет применить специальный способ вычисления, описанный выше. Вычислим вклад в меру глобального родства для каждого варианта.

Порядок (0-3-1):

cont(A_0, BUDGET, PNO) = 2*bond*(A_0, BUDGET) + 2*bond*(BUDGET, PNO) – 2*bond*(A_0, PNO).

Мы знаем, что

$bond(A_0, \texttt{PNO}) = bond(A_0, \text{BUDGET}) = 0;$
$bond(\texttt{BUDGET}, \texttt{PNO}) = 45 * 45 + 5 * 0 + 53 * 45 + 3 * 0 = 4410.$

Таким образом:

$cont(A_0, \texttt{BUDGET}, \texttt{PNO}) = 8820.$

	PNO	PNAME
PNO	45	0
PNAME	0	80
BUDGET	45	5
LOC	0	75

(a)

	PNO	PNAME	BUDGET
PNO	45	45	0
PNAME	0	5	80
BUDGET	45	53	5
LOC	0	3	75

(b)

	PNO	PNAME	BUDGET	LOC
PNO	45	45	0	0
PNAME	0	5	80	75
BUDGET	45	53	5	3
LOC	0	3	75	78

(c)

	PNO	PNAME	BUDGET	LOC
PNO	45	45	0	0
PNAME	45	53	5	3
BUDGET	0	5	80	75
LOC	0	3	75	78

(d)

Рис. 2.14 ❖ Вычисление кластерной матрицы родства (CA)

Порядок (1-3-2):

$$cont(\texttt{PNO}, \texttt{BUDGET}, \texttt{PNAME}) = 2bond(\texttt{PNO}, \texttt{BUDGET}) + 2bond(\texttt{BUDGET}, \texttt{PNAME}) - 2bond(\texttt{PNO}, \texttt{PNAME});$$
$bond(\texttt{PNO}, \texttt{BUDGET}) = bond(\texttt{BUDGET}, \texttt{PNO}) = 4410;$
$bond(\texttt{BUDGET}, \texttt{PNAME}) = 890;$
$bond(\texttt{PNO}, \texttt{PNAME}) = 225.$

Таким образом,

$cont(\texttt{PNO}, \texttt{BUDGET}, \texttt{PNAME}) = 10150.$

Порядок (2-3-4):

$$cont(\texttt{PNAME}, \texttt{BUDGET}, \texttt{LOC}) = 2bond(\texttt{PNAME}, \texttt{BUDGET}) + 2bond(\texttt{BUDGET}, \texttt{LOC}) - 2bond(\texttt{PNAME}, \texttt{LOC});$$
$bond(\texttt{PNAME}, \texttt{BUDGET}) = 890;$
$bond(\texttt{BUDGET}, \texttt{LOC}) = 0;$
$bond(\texttt{PNAME}, \texttt{LOC}) = 0.$

Таким образом,

$cont(\texttt{PNAME}, \texttt{BUDGET}, \texttt{LOC}) = 1780.$

Поскольку вклад порядка (1-3-2) наибольший, мы помещаем BUDGET справа от PNO (рис. 2.14b). Аналогичные вычисления для атрибута LOC показывают, что его следует поместить справа от PNAME (рис. 2.14c).

Наконец, строки переставляются в том же порядке, что столбцы, и окончательный результат показан на рис. 2.14d. ♦

На рис. 2.14d мы видим, что создано два кластера: тот, что находится в левом верхнем углу, содержит меньшие значения родства, а тот, что в правом нижнем, – большие. Эта кластеризация показывает, как следует расщепить атрибуты отношения PROJ. Но в общем случае граница, по которой проходит расщепление, не столь очевидна. Если матрица *CA* велика, то обычно образуется более двух кластеров и потенциальных вариантов секционирования несколько. Поэтому необходим более систематический подход к проблеме.

2.1.2.3. Алгоритм расщепления

Цель расщепления – найти такие наборы атрибутов, обращения к которым производятся из полностью или хотя бы по большей части различных наборов запросов. Например, если удастся найти два атрибута A_1 и A_2, к которым обращается только запрос q_1, и атрибуты A_3 и A_4, к которым обращаются только запросы q_2 и q_3, то определиться с фрагментами просто. Задача в том, чтобы придумать алгоритмический метод нахождения таких групп.

Рассмотрим кластерную матрицу атрибутов на рис. 2.15. Если зафиксировать точку на диагонали, то получится два набора атрибутов: $\{A_1, A_2, \ldots, A_i\}$ в левом верхнем углу (обозначенный *TA*) и $\{A_{i+1}, \ldots, A_n\}$ в правом нижнем углу (обозначенный *BA*).

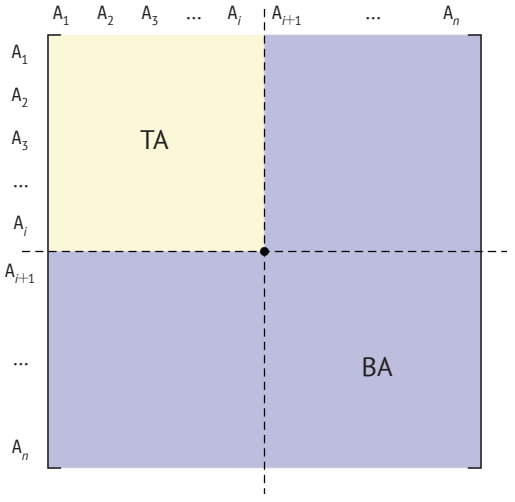


Рис. 2.15 ❖ Нахождение точки расщепления

Теперь разобьем все множество запросов $Q = \{q_1, q_2, \ldots, q_q\}$ на подмножества, обращающиеся только к атрибутам из *TA*, только из *BA* или к тем и другим. Эти подмножества определяются следующим образом:

$AQ(q_i) = \{A_j \mid use(q_i, A_j) = 1\}$;

$TQ = \{q_i \mid AQ(q_i) \subseteq TA\}$;
$BQ = \{q_i \mid AQ(q_i) \subseteq BA\}$;
$OQ = Q - \{TQ \cup BQ\}$.

Первое равенство определяет множество атрибутов, к которым обращается запрос q_i; TQ и BQ – множества запросов, обращающихся только к атрибутам из TA или BA соответственно, а OQ – множество запросов, обращающихся к тем и другим атрибутам.

Налицо задача оптимизации. Если в отношении n атрибутов, то существует $n - 1$ способов поставить точку деления на диагонали кластерной матрицы атрибутов. Лучшим будет тот, при котором порождаются такие множества TQ и BQ, для которых количество обращений только к *одному* фрагменту достигает максимума, а количество обращений к *обоим* фрагментам достигает минимума. Поэтому определим такие четыре величины:

$$CQ = \sum_{q_i \in Q} \sum_{\forall S_j} ref_i(q_i) acc_j(q_i);$$

$$CTQ = \sum_{q_i \in TQ} \sum_{\forall S_j} ref_i(q_i) acc_j(q_i);$$

$$CBQ = \sum_{q_i \in BQ} \sum_{\forall S_j} ref_i(q_i) acc_j(q_i);$$

$$COQ = \sum_{q_i \in OQ} \sum_{\forall S_j} ref_i(q_i) acc_j(q_i).$$

Каждая из них подсчитывает общее количество обращений к атрибутам из запросов, принадлежащих соответствующим классам. Тогда задача оптимизации состоит в том, чтобы найти точку x ($1 \leqslant x \leqslant n$), доставляющую максимум выражению

$$z = CTQ * CBQ - COQ^2.$$

Важная особенность этого выражения заключается в том, что оно определяет два фрагмента таких, что величины CTQ и CBQ настолько близки, насколько возможно. Это позволит сбалансировать нагрузку, когда фрагменты распределены по разным узлам. Ясно, что сложность алгоритма секционирования линейно зависит от количества атрибутов отношения, т. е. равна $O(n)$.

Эта процедура разбивает множество атрибутов на две части. Если множество атрибутов велико, то вполне может понадобиться m-путевое секционирование. Спроектировать такой алгоритм можно, но он будет вычислительно сложным. На диагонали матрицы CA нужно будет поставить 1, 2, ... , $m - 1$ точек разделения и для каждого количества точек найти позиции, при которых z достигает максимума. Таким образом, сложность алгоритма составляет $O(2^m)$. Конечно, определение z придется модифицировать для случая, когда точек разделения несколько. Альтернативное решение – рекурсивно применять алгоритм бинарного секционирования к каждому фрагменту, полученному на предыдущей итерации. Нужно будет вычислить TQ, BQ и OQ, а также ассоциированные с ними меры доступа для каждого фрагмента и продолжить их разбиение.

До сих пор мы предполагали, что точка деления одна, определяется однозначно и разбивает матрицу *CA* на левый верхний блок и второй блок, содержащий все остальные атрибуты. Однако разбиение можно сформировать и в середине матрицы. В этом случае алгоритм нужно немного модифицировать. Самый левый столбец матрицы *CA* сдвигается и становится самым правым, а верхняя строка сдвигается вниз. Операция сдвига сопровождается проверкой $n - 1$ положений на диагонали с целью нахождения максимального z. Идея такого сдвига заключается в том, чтобы переместить блок атрибутов, который должен составлять кластер, в левый верхний угол матрицы, где его легко найти. После добавления операции сдвига сложность алгоритма секционирования возрастает в n раз и оказывается равной $O(n^2)$.

В предположении, что процедура сдвига SHIFT уже реализована, алгоритм расщепления приведен в алгоритме 2.4. На входе он принимает кластерную матрицу родства *CA*, подлежащее фрагментации отношение, а также матрицы использования атрибутов и частот доступа. На выходе получается множество фрагментов $F_R = \{R_1, R_2\}$, где $R_i \subseteq \{A_1, A_2 \ldots, A_n\}$ и $R_1 \cap R_2$ = множество атрибутов, входящих в состав первичного ключа R. Отметим, что для n-путевого секционирования этот алгоритм нужно либо вызывать итеративно, либо реализовать как рекурсивную процедуру.

Алгоритм 2.4. SPLIT

Вход: *CA*: кластерная матрица родства; R: отношение; *ref*: матрица использования атрибутов; *acc*: матрица частот доступа
Выход: *F*: множество фрагментов
begin
    {определить значение *z* для первого столбца}
    {индексы в выражениях стоимости обозначают точку разделения}
    вычислить CTQ_{n-1}
    вычислить CBQ_{n-1}
    вычислить COQ_{n-1}
    $best \leftarrow CTQ_{n-1} * CBQ_{n-1} - (COQ_{n-1})^2$
    repeat
        {найти лучшее секционирование}
        for *i from n* − 2 *to* 1 *by* −1 **do**
            вычислить CTQ_i
            вычислить CBQ_i
            вычислить COQ_i
            $z \leftarrow CTQ * CBQ_i - COQ_i^2$
            if $z > best$ **then** $best \leftarrow z$ {запомнить точку разделения при данном сдвиге}
        end for
        call SHIFT(*CA*)
    until *не останется новых возможных сдвигов*
    реконструировать матрицу в соответствии со сдвигом
    $R_1 \leftarrow \prod_{TA}(R) \cup K$ {*K* – множество атрибутов, входящих в состав первичного ключа R}
    $R_2 \leftarrow \prod_{BA}(R) \cup K$
    $F \leftarrow \{R_1, R_2\}$
end

Пример 2.18. Применив алгоритм SPLIT к матрице *CA*, полученной для отношения PROJ (пример 2.17), мы найдем определение фрагментов F_{PROJ} = {$PROJ_1$, $PROJ_2$}, где

$PROJ_1$ = {PNO, BUDGET}
$PROJ_2$ = {PNO, PNAME, LOC}

Заметим, что в этом упражнении мы производили фрагментацию по всему множеству атрибутов, а не только по неключевым. Это сделано исключительно для того, чтобы не усложнять пример. Поэтому мы включили атрибут PNO, являющийся ключом PROJ как в $PROJ_2$, так и в $PROJ_1$. ♦

2.1.2.4. Проверка корректности

Мы будем рассуждать так же, как в случае горизонтальной фрагментации, чтобы доказать, что алгоритм SPLIT дает корректную вертикальную фрагментацию.

Полнота

Алгоритм SPLIT гарантирует полноту, потому что каждый атрибут глобального отношения помещен в один из фрагментов. При условии что множество атрибутов, над которым определено отношение R, имеет вид $A = \cup R_i$, полнота вертикальной фрагментации доказана.

Реконструкция

Мы уже упоминали, что реконструировать исходное глобальное отношение можно с помощью операции соединения. Таким образом, для отношения R с вертикальной фрагментацией $F_R = \{R_1, R_2, \ldots, R_r\}$ и ключевыми атрибутами K имеет место соотношение $R = \bowtie_K R_i$, $\forall R_i \in F_R$. Поэтому при условии, что каждый R_i полон, операция соединения правильно реконструирует R. Есть еще один важный момент – каждый фрагмент R_i должен либо содержать атрибуты R, входящие в состав первичного ключа, либо назначенный системой идентификатор кортежа (TID).

Дизъюнктность

Как уже было сказано, атрибуты первичного ключа повторяются в каждом фрагменте. Но если их не считать, то алгоритм SPLIT находит взаимно непересекающиеся кластеры атрибутов, а значит, соответствующие им фрагменты дизъюнктны.

2.1.3. Гибридная фрагментация

В некоторых случаях простой горизонтальной или вертикальной фрагментации схемы базы данных недостаточно, чтобы удовлетворить потребности приложений. В таком случае вертикальную фрагментацию можно затем дополнить горизонтальной, или наоборот, так что порождается древовидное секционирование (рис. 2.16). Поскольку разные стратегии секционирования

применяются последовательно – сначала одна, потом другая, – этот подход называется гибридной фрагментацией. Встречаются также прилагательные *смешанная* и *вложенная*.

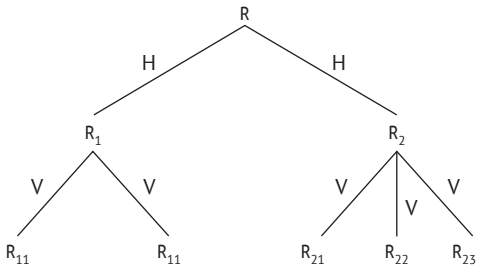


Рис. 2.16 ❖ Гибридная фрагментация

Хороший пример, доказывающий необходимость гибридной фрагментации, дает отношение `PROJ`. В примере 2.10 мы разбили его на шесть горизонтальных фрагментов, исходя из поведения двух приложений. В примере 2.18 то же самое отношение мы разбили на два вертикальных фрагмента. Таким образом, мы имеем множество горизонтальных фрагментов, каждый из которых затем разбит на два вертикальных.

Правила корректности и условия для гибридной фрагментации естественно следуют из правил и условий для вертикальной и горизонтальной фрагментации. Например, чтобы реконструировать исходное глобальное отношение в случае гибридной фрагментации, нужно начать с листьев дерева фрагментации и подниматься вверх, выполняя операции соединения и объединения (рис. 2.17). Фрагментация полна, если полны листовые и промежуточные фрагменты. Аналогично дизъюнктность гарантируется, если листовые и промежуточные фрагменты являются дизъюнктными.

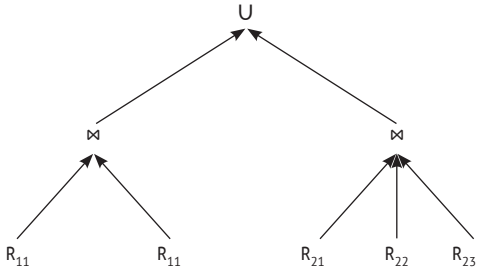


Рис. 2.17 ❖ Реконструкция гибридной фрагментации

2.2. Размещение

После фрагментации нужно решить следующую проблему – разместить фрагменты в узлах распределенной СУБД. Мы можем либо поместить каждый

фрагмент в одном узле, либо реплицировать его на несколько узлов. Для репликации есть две причины: надежность и эффективность чтения. Если имеется несколько копий фрагмента, то велика вероятность, что хотя бы одна копия останется доступной даже в случае отказа системы. Кроме того, запросы на чтение данных можно выполнять параллельно, поскольку копии размещены в нескольких узлах. С другой стороны, обновление данных вызывает проблемы, потому что система должна гарантировать правильное обновление всех копий данных. Поэтому решение является результатом компромисса и зависит от соотношения запросов на чтение и обновление. Это решение так или иначе отражается почти на всех алгоритмах и функциях управления распределенными СУБД.

Нереплицированная база данных (называемая также *секционированной*) содержит фрагменты, размещенные так, что каждый фрагмент находится только в одном узле. В случае репликации либо база данных целиком хранится в каждом узле (*полностью реплицированная* база данных), либо фрагменты распределены по узлам так, что копии фрагмента могут находиться в нескольких узлах (*частично реплицированная* база данных). В последнем случае количество копий фрагмента может подаваться на вход алгоритма или вычисляться самим алгоритмом. На рис. 2.18 эти три варианта репликации сопоставляются с другими функциями распределенной СУБД. Подробно мы будем обсуждать репликацию в главе 6.

	Полная репликация	Частичная репликация	Секционирование
ОБРАБОТКА ЗАПРОСОВ	Легко	Одинаково трудно	
УПРАВЛЕНИЕ КАТАЛОГОМ	Легко или не требуется	Одинаково трудно	
УПРАВЛЕНИЕ КОНКУРЕНТНОСТЬЮ	Умеренно	Трудно	Легко
НАДЕЖНОСТЬ	Очень высокая	Высокая	Низкая
РЕАЛИСТИЧНОСТЬ	Для некоторых приложений	Реалистично	Для некоторых приложений

Рис. 2.18 ❖ Сравнение вариантов репликации

Проблема размещения файлов (ПРФ, англ. FAP – file allocation problem) долго изучалась в контексте распределенных вычислительных систем, в которых единицей размещения является файл. Ее постановка обычно очень проста, что отражает простоту API для работы с файлами. Но даже в таком простом варианте эта задача является NP-полной, поэтому приходится искать разумные эвристики.

Постановка ПРФ не годится для проектирования распределенной базы данных в силу фундаментальных характеристик СУБД: фрагменты не являются независимыми, поэтому их нельзя просто так отобразить на отдельные файлы; доступ к данным в базе устроен сложнее, чем доступ к файлам. Кроме того, СУБД обязаны гарантировать целостность и свойства транзакционности, и затраты на это необходимо учитывать.

Не существует общей эвристической модели, которая принимала бы на входе множество фрагментов и порождала близкое к оптимальному размещение при соблюдении разного рода ограничений. В современных моделях делается ряд упрощающих предположений, и применимы они лишь в определенных обстоятельствах. Поэтому вместо того чтобы представлять алгоритмы размещения, мы опишем общую модель, а затем обсудим несколько возможных эвристик для ее реализации.

2.2.1. Дополнительная информация

Нам необходимы количественные сведения о базе данных, рабочей нагрузке, сети связи, возможностях процессоров и ограничениях по емкости хранения в каждом узле сети.

Для выполнения горизонтальной фрагментации мы определили избирательность элементарных конъюнкций. Теперь нужно расширить определение фрагментов и определить избирательность фрагмента F_j относительно запроса q_i. Так называется число кортежей F_j, к которым необходимо обратиться для обработки q_i. Будем обозначать эту величину $sel_i(F_j)$.

Еще одна важная информация о фрагментах базы данных – их размер. Размер фрагмента F_j определяется по формуле

$$size(F_j) = card(F_j) * length(F_j),$$

где $length(F_j)$ – длина кортежа фрагмента F_j в байтах.

Большая часть относящейся к рабочей нагрузке информации уже учтена в процессе фрагментации, но для модели размещения нужно кое-что еще. Два важных показателя – количество операций чтения фрагмента F_j во время выполнения запроса q_i (обозначается RR_{ij}) и аналогично количество операций обновления (UR_{ij}). Например, можно подсчитать количество блоков, к которым производится доступ со стороны запроса.

Нам также понадобится определить две матрицы *UM* и *RM* с элементами u_{ij} и r_{ij} соответственно:

$$u_{ij} = \begin{cases} 1, \text{ если запрос } q_i \text{ обновляет фрагмент } F_j, \\ 0 \text{ в противном случае;} \end{cases}$$

$$r_{ij} = \begin{cases} 1, \text{ если запрос } q_i \text{ читает фрагмент } F_j, \\ 0 \text{ в противном случае.} \end{cases}$$

Также определим вектор *O*, состоящий из элементов $o(i)$, описывающих, в каком узле был предъявлен запрос q_i. Наконец, необходимо задать максимальное время ожидания ответа каждым приложением.

Для каждого вычислительного узла нужно знать емкость его системы хранения и обрабатывающие возможности. Эти величины можно получить в результате сложных вычислений или в виде простых оценок. Стоимость единицы хранения данных в узле S_k обозначим USC_k. Еще нужно задать по-

казатель LPC_k, равный стоимости обработки одной единицы работы в узле S_k. Единица измерения работы должна быть такой же, как в показателях *RR* и *UR*.

В нашей модели предполагается существование простой сети, в которой стоимость коммуникации определяется в терминах одного сообщения, содержащего определенный объем данных. Обозначим g_{ij} стоимость связи в расчете на одно сообщение между узлами S_i и S_j. Чтобы посчитать количество сообщений, положим размер одного сообщения в байтах равным *msize*. Существуют более развитые модели сети, в которых учитывается пропускная способность каналов, расстояние между узлами, накладные расходы протоколов и т. д., но для наших целей хватит и этой простой модели.

2.2.2. Модель размещения

Мы рассматриваем модель размещения, которая стремится минимизировать полную стоимость обработки и хранения, соблюдая ограничения на время ответа. Модель имеет вид

min(Total Cost)

при условии

ограничений на время ответа,
ограничений на емкость системы хранения,
ограничений на мощность процессоров.

Далее в этом разделе мы раскроем составные части этой модели, основываясь на требованиях к информации, обсуждавшихся в разделе 2.2.1. Переменная решения x_{ij} определяется следующим образом:

$$x_{ij} = \begin{cases} 1, \text{ если фрагмент } F_i \text{ хранится в узле } S_j, \\ 0 \text{ в противном случае.} \end{cases}$$

2.2.2.1. Полная стоимость

Функция полной стоимости состоит из двух частей: обработки запроса и хранения. Поэтому ее можно записать в виде

$$TOC = \sum_{\forall q_i \in Q} QPC_i + \sum_{\forall S_k \in S} \sum_{\forall F_j \in F} STC_{jk},$$

где QPC_i – стоимость обработки запроса q_i, а STC_{jk} – стоимость хранения фрагмента F_j в узле S_k.

Сначала рассмотрим стоимость хранения. Она вычисляется по формуле

$$STC_{jk} = USC_k * size(F_j) * x_{jk},$$

а суммирование производится для нахождения полной стоимости по всем узлам и всем фрагментам.

Описать стоимость обработки запроса труднее. Мы будем считать, что она складывается из стоимости обработки (*PC*) и стоимости передачи информации (*TC*). Таким образом, стоимость обработки запроса (*QPC*) со стороны приложения q_i равна

$$QPCi = PC_i + TC_i.$$

Часть *PC* складывается из трех элементов – стоимости доступа (*AC*), стоимости гарантирования целостности (*IE*) и стоимости управления конкурентностью (*CC*):

$$PC_i = AC_i + IE_i + CC_i.$$

Детальная спецификация каждого элемента зависит от алгоритмов, применяемых для выполнения соответствующих задач. Для демонстрации распишем *AC* более подробно:

$$AC_i = \sum_{\forall S_k \in S} \sum_{\forall F_j \in F} (u_{ij} * UR_{ij} + r_{ij} * RR_{ij}) * x_{jk} * LPC_k.$$

Первые два члена в этой формуле вычисляют количество обращений запроса q_i к фрагменту F_j. Заметим, что ($UR_{ij} + RR_{ij}$) дает полное число операций обновления и выборки. Мы предполагаем, что локальные стоимости их обработки одинаковы. Суммирование дает полное число операций доступа ко всем фрагментам, к которым обращается q_i. Умножение на LPC_k дает стоимость этих операций доступа в узле S_k. Мы снова используем x_{jk}, чтобы выбирать стоимости только для узлов, где хранятся фрагменты.

Функция стоимости доступа предполагает, что обработка запроса включает разложение его на множество подзапросов, каждый из которых работает над фрагментом, хранящимся в узле, а затем передачу результатов узлу, который инициировал запрос. Реальность сложнее; например, эта функция стоимости не учитывает затраты на выполнение соединений (при необходимости), которое можно реализовать разными способами (см. главу 4).

Стоимость гарантирования целостности можно описать как стоимость обработки, только единичная стоимость локальной обработки, скорее всего, изменится, чтобы отразить истинные затраты на поддержание целостности. Поскольку методы контроля целостности и управления конкурентностью обсуждаются ниже в этой книге, мы не будем вдаваться в эти детали прямо сейчас. Читателю рекомендуется вернуться к этому разделу после прочтения глав 3 и 5 и убедиться, что функции стоимости действительно можно вывести.

Функцию стоимости передачи можно описать примерно так же, как функцию стоимости доступа. Однако накладные расходы на передачу данных при обновлении и при выборке сильно различаются. Для запроса обновления необходимо проинформировать все узлы, на которых присутствуют реплики, тогда как для запроса выборки достаточно обратиться всего к одной копии. Кроме того, по завершении запроса обновления не нужно передавать узлу, инициировавшему запрос, никакие данные, кроме подтверждения, а в случае выборки, возможно, придется передать большой объем данных.

Компонента функции стоимости передачи, связанная с обновлением, имеет вид

$$TCU_i = \sum_{\forall S_k \in S} \sum_{\forall F_j \in F} u_{ij} * x_{jk} * g_{o(i),k} + \sum_{\forall S_k \in S} \sum_{\forall F_j \in F} u_{ij} * x_{jk} * g_{k,o(i)}.$$

Первый член относится к отправке сообщения об обновлении от узла $o(i)$, инициировавшего запрос q_i, всем репликам фрагментов, подлежащим обновлению. Второй член относится к подтверждению.

Стоимость выборки можно записать в виде

$$TCR_i = \sum_{\forall F_j \in F} \min_{S_k \in S} \left(r_{ij} * x_{jk} * g_{o(i),k} + r_{ij} * x_{jk} * \frac{sel_i(F_j) * length(F_j)}{msize} * g_{k,o(i)} \right).$$

Первый член *TCR* представляет стоимость передачи запроса выборки тем узлам, на которых имеются копии фрагментов с нужными данными. Второй член учитывает передачу результатов от этих узлов узлу-инициатору. Эта формула означает, что среди всех узлов, где хранятся копии фрагмента, для выполнения операции выбирается тот, для которого полная стоимость передачи минимальна.

Теперь полную функцию стоимости передачи для запроса q_i можно записать так:

$$TC_i = TCU_i + TCR_i.$$

2.2.2.2. Ограничения

Функции ограничений можно описать с той же степенью детальности. Но мы вместо этого просто покажем, как они должны выглядеть. Ограничение на время ответа должно быть записано в виде:

время выполнения $q_i \leqslant$ максимальное время выполнения q_i, $\forall q_i \in Q$.

Желательно выражать стоимость в целевой функции в терминах времени, поскольку тогда спецификация ограничения на время выполнения получается относительно простой.

Ограничение на хранение имеет вид:

$$\sum_{\forall F_i \in F} STC_{jk} \leq \text{емкость хранения в узле } S_k,\ \forall S_k \in S,$$

а ограничение на мощность процессоров имеет вид:

$$\sum_{\forall q_i \in Q} \begin{matrix} \text{нагрузка на процессор, создаваемая запросом } q_i \\ \text{в узле } S_k \end{matrix} \leq \text{мощность процессоров } S_k,\ \forall S_k \in S.$$

На этом мы завершаем разработку модели размещения. Хотя мы и не довели ее до конца, детальность проработки некоторых элементов показывает, как подходить к постановке такой задачи. Кроме того, мы отметили некоторые важные вопросы, которые следует решить в моделях размещения.

2.2.3. Методы решения

Как уже было сказано, даже простая проблема размещения файлов является NP-полной. Поскольку модель, разработанная нами в предыдущем разделе, сложнее, вероятно, она тоже NP-полная. Поэтому нужно искать эвристические методы, которые дают неоптимальные решения. Критерием «качества» в данном случае, конечно, является близость результатов эвристического алгоритма к оптимальному размещению.

Довольно давно была замечена связь между задачами о размещении файлов и о размещении объектов. На самом деле можно установить изоморфизм между проблемой размещения файлов и задачей размещения складов одного товара. Поэтому эвристические методы решения второй применялись и для решения первой. В качестве примеров приведем решение задачи о рюкзаке, метод ветвей и границ и алгоритмы потоков в сетях.

Были и другие попытки уменьшить сложность задачи. Одна из стратегий заключается в предположении о том, что все потенциальные фрагментации вместе с ассоциированными стоимостями и преимуществами определены в терминах обработки запросов. Тогда задача моделируется как выбор оптимальной фрагментации и размещения каждого отношения. Другое часто используемое упрощение – сначала игнорировать репликацию и найти оптимальное нереплицированное решение. Репликация же учитывается на втором этапе путем применения жадного алгоритма, который в качестве начального приближения берет нереплицированное решение и пытается улучшить его. Но для этих эвристик не хватает данных, чтобы определить, насколько результаты близки к оптимальным.

2.3. Комбинированные подходы

В процессе проектирования, изображенном на рис. 2.2, который мы положили в основу обсуждения, шаги фрагментации и размещения разделены. Методика последовательная – результат фрагментации подается на вход размещения; мы называем ее *фрагментируй-затем-размещай*. При таком подходе постановка задачи упрощается, поскольку уменьшается пространство решений, но в действительности разделение на два шага может увеличить сложность моделей размещения. Входные данные для обоих шагов похожи, а отличие только в том, что фрагментация применяется к глобальным отношениям, а размещение – к фрагментам. В обоих случаях нужна информация о рабочей нагрузке, но вопрос о том, как она используется на другом шаге, игнорируется. В результате алгоритмы фрагментации решают, как секционировать отношение, частично опираясь на информацию о том, как к ней обращаются запросы, но модели размещения игнорируют тот факт, что эта информация играет роль при фрагментации. Поэтому модели размещения должны заново включать все детали связей между фрагментами отношений и информацию о том, как к ним обращаются приложения. Существуют подходы, в которых шаги фрагментации и размещения объединены таким об-

разом, что алгоритм секционирования данных диктует их размещение или же алгоритм размещения диктует способ секционирования; мы называем их *комбинированными подходами*. По большей части они относятся к горизонтальному секционированию, потому что это общий способ достижения значительного параллелизма. В этом разделе мы представим эти подходы, которые можно отнести к двум группам: безразличные к рабочей нагрузке и учитывающие рабочую нагрузку.

2.3.1. Методы секционирования, безразличные к рабочей нагрузке

Методы этого класса игнорируют характер рабочей нагрузки на данные, а смотрят только на саму базу данных, часто даже не обращая внимания на определение схемы. Они обычно используются в параллельных СУБД, где динамичность данных выше, чем в распределенных СУБД, поэтому предпочтительны более простые и быстрые методы.

Простейшая форма таких алгоритмов – *циклическое секционирование* (рис. 2.19). Если имеется n секций, то i-й вставляемый кортеж назначается секции с номером ($i \bmod n$). Эта стратегия позволяет параллельно осуществлять последовательный доступ к отношению. Однако для прямого доступа к отдельным кортежам по предикату придется работать со всем отношением. Поэтому циклическое секционирование подходит для запросов с полным просмотром, как в задачах добычи данных.

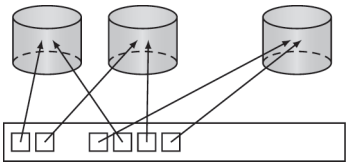


Рис. 2.19 ❖ Циклическое секционирование

Альтернативой является *хеш-секционирование*, когда к некоторому атрибуту применяется хеш-функция, возвращающая номер секции (рис. 2.20). Эта стратегия позволяет выполнять запросы по точному совпадению с выбранным атрибутом ровно на одном узле, а все прочие запросы – параллельно на всех узлах. Однако если распределение атрибута, выбранного для секционирования, неравномерно, как, например, в случае фамилий людей, то получающееся размещение может оказаться несбалансированным, т. е. некоторые секции будут гораздо больше других. Этот важный аспект называется *асимметрией данных* и может привести с дисбалансу нагрузки.

Наконец, в случае секционирования по диапазонам (рис. 2.21) кортежи распределяются по интервалам значений (диапазонам) некоторого атрибута, что позволяет решить проблему неравномерности распределения данных. В отличие от хеширования, опирающегося на хеш-функции, диапазоны не-

обходимо хранить в индексной структуре, например B-дереве. Этот метод хорошо подходит не только для запросов по точному совпадению (как в случае хеширования), но и для запросов по диапазону. Например, запрос с предикатом «A between A_1 and A_2» можно обработать только в узлах, содержащих кортежи, в которых значение атрибута A принадлежит диапазону $[A_1, A_2]$.

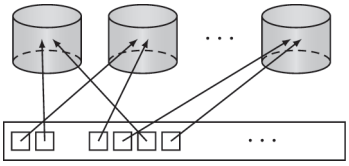


Рис. 2.20 ❖ Хеш-секционирование

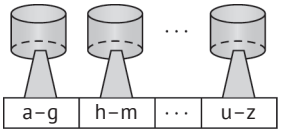


Рис. 2.21 ❖ Секционирование по диапазонам

Эти методы просты, не требуют длительных вычислений и, как станет ясно в главе 8, отлично согласуются с динамичностью данных в параллельных СУБД. Они также косвенно поддерживают семантические связи между отношениями в базе данных. Например, рассмотрим два отношения, между которыми имеется соединение по совпадению первичного и внешнего ключей, $\mathtt{R} \bowtie_{\mathtt{R.A=S.B}} \mathtt{S}$. Тогда если при хеш-секционировании использовать одну и ту же функцию для атрибутов `R.A` и `S.B`, то они гарантированно попадут в один и тот же узел, в результате чего соединение будет локализовано и его можно будет вычислить параллельно. Аналогичный подход применим к секционированию по диапазонам, но при циклическом секционировании эта связь не принимается во внимание.

2.3.2. Методы секционирования, учитывающие рабочую нагрузку

Методы этого класса рассматривают рабочую нагрузку как исходные данные и производят секционирование, так чтобы по возможности локализовать нагрузку на одном узле. Как было сказано в начале этой главы, их цель – минимизировать количество распределенных запросов.

Один такой подход был предложен в системе Schism, где информация о базе данных и рабочей нагрузке используется для построения графа $G = (V, E)$, в котором каждая вершина $v \in V$ представляет кортеж базы данных, а каждое ребро $e = (v_i, v_j) \in E$ – запрос, обращающийся к обоим кортежам v_i и v_j. Каждому ребру назначается вес, равный количеству транзакций, обращающихся к обоим кортежам.

В этой модели легко также учесть реплики, если представить каждую копию отдельной вершиной. Количество вершин, соответствующих репликам, определяется числом транзакций, обращающихся к кортежу, т. е. каждая транзакция обращается к одной копии. Реплицированный кортеж представлен в графе звездообразной структурой, состоящей из $n + 1$ вершин, в которой «центральная» вершина соответствует логическому кортежу, а остальные n – его физическим копиям. Вес ребра между вершиной физической копии и центральной вершиной равен количеству транзакций, обновляющих кортеж; веса остальных ребер по-прежнему равны количеству транзакций, обращающихся к кортежу. Такая организация имеет смысл, потому что наша цель – по возможности локализовать транзакции, а в этом методе репликация используется, чтобы достичь локализации.

Пример 2.19. Рассмотрим базу данных с одним отношением, содержащим семь кортежей, к которым обращаются пять транзакций. На рис. 2.22 изображен построенный граф: имеется семь вершин, соответствующих кортежам, а обращающиеся к ним запросы собраны в клики. Например, запрос Q_1 обращается к кортежам 2 и 7, запрос Q_2 – к кортежам 2, 3 и 6, запрос Q_3 – к кортежам 1, 2 и 3, запрос Q_4 – к кортежам 3, 4 и 5, а запрос Q_5 – к кортежам 4 и 5. Вес каждого ребра равен количеству обращающихся транзакций.

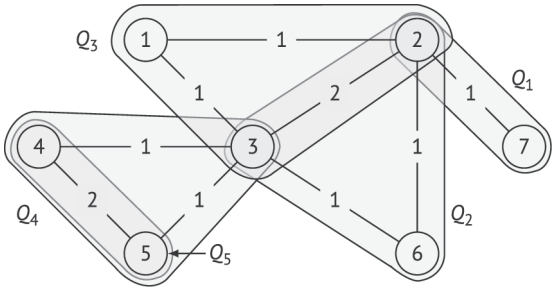


Рис. 2.22 ❖ Представление графа для секционирования в Schism

В этот граф можно включить репликацию, размножив кортежи, к которым обращается несколько транзакций, это показано на рис. 2.23. Заметим, что кортежи 1, 6 и 7 не реплицированы, потому что к каждому из них обращается только одна транзакция, кортежи 4 и 5 реплицированы дважды, а кортежи 2 и 3 – трижды. «Ребра репликации» между центральной вершиной и физическими копиями представлены штриховыми линиями, а их веса в данном примере опущены. ♦

После того как база данных и рабочая нагрузка отражены в виде графа, следующий шаг – произвести разрезание графа. Поскольку эта техника подробно обсуждается в разделе 10.4.1, мы сейчас не станем вдаваться в детали, а просто скажем, что в результате разрезания графа каждая вершина включается в некоторую секцию графа, так что никакие две секции не имеют общих вершин. Цель алгоритмов этого вида – найти сбалансированное (или почти сбалансированное) множество секций, минимизировав при этом стоимость

реберных разрезов. Стоимость реберных разрезов принимает во внимание веса всех ребер, т. е. в результате минимизируется количество распределенных запросов.

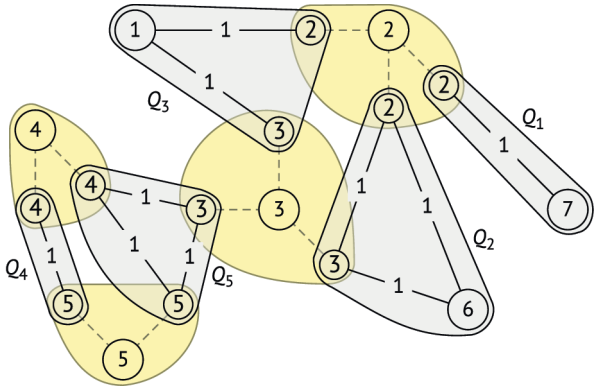


Рис. 2.23 ❖ Граф в Schism, включающий информацию о репликации

Преимущество подхода, принятого в Schism, в детальности размещения – каждый кортеж рассматривается как единица размещения, а секционирование «возникает», когда принимается решение о размещении каждого кортежа. Поэтому отображением множеств кортежей на запросы можно управлять, и многие запросы будут выполняться в одном узле. Но есть и недостаток – по мере роста базы данных и, в частности, при добавлении в граф реплик размер графа непомерно увеличивается. Поэтому управление графом усложняется, и секционирование становится слишком дорогим. Кроме того, таблицы отображения, в которых запоминается местоположение каждого кортежа (т. е. каталог), также оказываются очень велики, и управление ими превращается в самостоятельную проблему.

Один из способов преодолеть эти проблемы был предложен в системе SWORD, где используется гиперграфовая модель[1], в которой каждая клика на рис. 2.22 представлена гиперребром. Каждое гиперребро представляет один запрос, а множество вершин, соединенных гиперребром, – кортежи, к которым он обращается. У каждого гиперребра имеется вес, представляющий частоту этого запроса в рабочей нагрузке. Таким образом, мы имеем взвешенный гиперграф. Затем этот гиперграф разрезается с помощью алгоритма k-путевого минимального разреза, который порождает k сбалансированных секций, каждой из которых назначается узел. В результате количество распределенных разрезов минимизируется, потому что алгоритм минимизирует число разрезов гиперребер, а каждый из разрезов соответствует распределенному запросу.

[1] В гиперграфе каждое ребро (называемое гиперребром) может соединять не две вершины, как в обычном графе, а больше. Детали теории гиперграфов выходят за рамки этой книги.

Разумеется, этого изменения модели недостаточно для решения вышеупомянутых проблем. Чтобы уменьшить размер графа и накладные расходы на поддержание таблицы отображения, SWORD сжимает гиперграф следующим образом. Множество вершин V исходного гиперграфа G отображается в множество виртуальных вершин V' с помощью хеш-функции или другой функции, которая применяется к первичным ключам кортежей. После того как множество виртуальных вершин определено, гиперребра исходного гиперграфа отображаются на гиперребра сжатого графа (E'), так что если вершины, принадлежащие гиперребру $e \in E$, отображены на разные виртуальные вершины сжатого графа, то существует гиперребро $e' \in E'$.

Конечно, чтобы такое сжатие имело смысл, должно быть $|V'| < |V|$, поэтому главная проблема – решить, какая степень сжатия желательна; если она слишком велика, то количество виртуальных вершин уменьшится, но увеличится количество гиперребер, а вместе с ним и возможность появления распределенных запросов. Результирующий сжатый гиперграф $G' = (V', E')$ будет меньше исходного, и, стало быть, управлять им и разрезать его будет проще, и таблицы отображения также уменьшатся, поскольку в них хранятся только виртуальные вершины.

Пример 2.20. Вернемся к примеру 2.19 и будем считать, что гиперграф сжимается в три виртуальные вершины $v'_1 = 1, 2$, $v'_2 = 3, 4, 5$, $v'_3 = 6, 7$. Тогда будет два гиперребра: $e'_1 = (v'_1, v'_3)$ с частотой 2 (соответствует ребрам Q_1 и Q_2 исходного гиперграфа) и $e'_2 = (v'_1, v'_2)$ с частотой 1 (соответствует Q_3). Гиперребра, представляющие запросы Q_4 и Q_5, локальны (т. е. не охватывают виртуальные вершины), поэтому в сжатом графе они не нужны. Результат показан на рис. 2.24. ♦

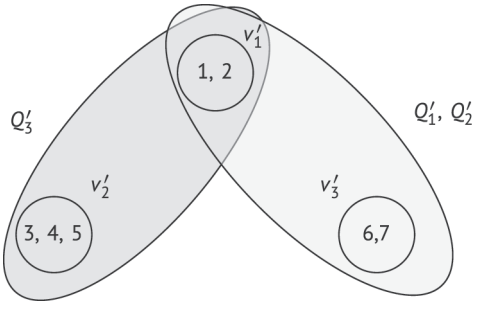


Рис. 2.24 ❖ Сжатый гиперграф в Sword

Произвести k-путевое сбалансированное минимальное разрезание сжатого гиперграфа можно гораздо быстрее, а получающаяся таблица отображения будет меньше благодаря уменьшению размера графа.

SWORD учитывает репликацию в сжатом графе. Сначала для каждой виртуальной вершины определяется, сколько необходимо реплик. Для этого используется статистика доступа на уровне кортежей для каждого кортежа t_j в каждой виртуальной вершине v'_i, а именно частота его чтения f^r_{ij} и записи f^w_{ij}. Зная эти величины, можно вычислить средние частоты чтения и записи (соответственно ARF и AWF) виртуальной вершины v'_i по формулам:

$$ARF(v_i') = \frac{\sum_j f_{ij}^r}{\log S(v_i')} \quad \text{и} \quad AWF(v_i') = \frac{\sum_j f_{ij}^w}{\log S(v_i')},$$

где $S(v_i')$ – размер каждой виртуальной вершины (в терминах количества отображенных на нее реальных вершин), а логарифм берется, чтобы компенсировать асимметрию размеров виртуальных вершин (так что это средние с поправкой на размер). По ним SWORD определяет коэффициент репликации $\mathcal{R} = \frac{AWF(v_i')}{ARWF(v_i')}$, а пользователь определяет пороговую величину δ ($0 < \delta < 1$). Количество реплик (*#_rep*) виртуальной вершины v_i' теперь определяется следующим образом:

$$\#\_rep(v_i') = \begin{cases} 1, & \text{если } \mathcal{R} \geq \delta \\ ARF(v_i') & \text{в противном случае} \end{cases}.$$

После того как количество реплик каждой виртуальной вершины определено, они добавляются в сжатый гиперграф и распределяются по гиперребрам, так чтобы минимизировать минимальный разрез в алгоритме разрезания. Как именно производится это распределение, мы описывать не станем.

2.4. Адаптивные подходы

Излагая материал этой главы, мы, вообще говоря, предполагали статическую среду, в которой проектирование выполняется один раз, после чего найденная структура сохраняет актуальность. Но, конечно, реальность выглядит совершенно иначе. Физические (например, характеристики сети, располагаемая емкость хранения в каждом узле) и логические (рабочая нагрузка) изменения вынуждают перепроектировать базу данных. В динамической среде процесс принимает вид проектирование–перепроектирование–материализация нового проекта. Когда что-то изменяется, проще всего переработать проект распределения с нуля. Но для больших или особо динамичных систем это не всегда возможно, потому что накладные расходы на перепроектирование, скорее всего, будут очень высоки. Предпочтительнее выполнять перепроектирование инкрементно, обращая внимание только на те части базы данных, которые с высокой вероятностью оказываются затронуты изменениями. Инкрементное перепроектирование можно выполнять при каждом обнаружении изменения или периодически, по мере того как изменения накапливаются и с регулярными интервалами оцениваются.

Основные работы в этой области сосредоточены на изменениях рабочей нагрузки (запросы и транзакции) со временем, и именно сюда мы направим внимание в этом разделе. Хотя некоторые работы в этом направлении ориентированы на подход «фрагментируй-затем-размещай», большинство следует комбинированной стратегии. В первом случае была, в частности, предложена идея, включающая фазу расщепления, на которой фрагменты подвергаются более мелкому разбиению на основе изменившихся требований к приложе-

нию – и так до тех пор, пока измельчение не перестанет приносить выигрыш в соответствии с некоторой функцией стоимости. В этот момент начинается фаза слияния, на которой фрагменты, к которым производится совместное обращение со стороны набора приложений, объединяются в один. Но нас будут больше интересовать динамические комбинированные подходы, при которых инкрементное перепроектирование производится по мере изменения рабочей нагрузки.

Цель адаптивных подходов такая же, как у стратегий секционирования, безразличных к рабочей нагрузке, рассмотренных в разделе 2.3.2: минимизировать количество распределенных запросов и гарантировать, что данные, обрабатываемые каждым запросом, локальны. В этом контексте имеется три взаимосвязанных вопроса, на которые нужно ответить при адаптивном проектировании распределения.

1. Как обнаружить изменения рабочей нагрузки, требующие внесения изменений в проект распределения?
2. Как определить, на какие данные повлияет перепроектирование?
3. Как эффективно осуществить изменения?

Далее мы обсудим эти вопросы.

2.4.1. Обнаружение изменений рабочей нагрузки

Это трудный вопрос, и работ на эту тему не так много. Авторы большинства предложенных адаптивных методов предполагают, что изменение рабочей нагрузки уже выявлено, и сразу переходят к проблеме миграции. Чтобы обнаружить изменение рабочей нагрузки, нужно наблюдать за поступающими запросами. Один из способов – периодически анализировать системные журналы, но с этим могут быть связаны значительные накладные расходы, особенно в высокодинамичных системах. Альтернатива – вести постоянный мониторинг рабочей нагрузки внутри самой СУБД. Вышеупомянутая система SWORD отслеживает процентную долю числа распределенных транзакций и считает, что система изменилась настолько, что требуется реконфигурация, если увеличение этой доли превысило заданный порог. С другой стороны, система E-Store ведет наблюдение за системными метриками и доступом на уровне кортежей. Сначала собираются системные метрики в каждом вычислительном узле с помощью средств ОС. В текущей версии E-Store в первую очередь стремится обнаружить несбалансированность нагрузки на разных узлах, а потому собирает только данные об использовании процессоров. Если дисбаланс потребления процессорного времени превышает порог, то запускается более детальный мониторинг на уровне кортежей, чтобы найти проблемные участки (см. следующий раздел). Хотя несбалансированное использование процессоров может являться хорошим индикатором проблем с производительностью, этот способ слишком прост для улавливания значительных изменений рабочей нагрузки. Конечно, можно организовать более сложный мониторинг, например создать профиль, в котором запоминаются частоты всех запросов на протяжении заданного промежутка времени, доля запросов, не выходящих (или выходящих) за пределы оговоренных задержек

(скажем, указанных в соглашении об уровне обслуживания), и прочее. Затем можно решить, достаточно ли велики изменения в профиле, чтобы приступать к перепроектированию, и делать это можно как непрерывно (т. е. всякий раз, как монитор регистрирует информацию), так и периодически. Проблема в том, как сделать это эффективно, не снижая производительность системы. Это область активных исследований, пока еще недостаточно изученная.

2.4.2. Обнаружение проблемных участков

После того как изменение рабочей нагрузки обнаружено, наступает время следующего шага – определить, какие данные им затронуты и должны быть перенесены в другое место. Как это делается, во многом зависит от метода обнаружения. Например, если система ведет мониторинг частоты запросов и обнаруживает изменения, то сами запросы говорят о данных. Можно обобщить наблюдения и перейти к шаблонам запросов, чтобы выявить «похожие» запросы, на которые изменения также могли повлиять. Это сделано в системе Apollo, в которой все константы заменены подстановочными символами. Например, запрос

```
SELECT PNAME FROM PROJ WHERE BUDGET>200000 AND LOC = "London"
```

был бы обобщен следующим образом:

```
SELECT PNAME FROM PROJ WHERE BUDGET>? AND LOC = "?"
```

При этом, конечно, расширяется точное множество данных, затронутых изменениями, но зато могут быть найдены дополнительные данные, на которые могут повлиять похожие запросы, и, значит, перепроектировать систему придется не так часто.

Система E-Store запускает мониторинг на уровне кортежей, после того как обнаружит несбалансированную загрузку системы. В течение короткого времени она собирает данные о доступе к кортежам в каждом вычислительном узле (т. е. в каждой секции) и выявляет «горячие» кортежи, т. е. те, к которым было больше всего обращений за это время. С этой целью для каждого кортежа используется гистограмма, которая инициализируется в момент активации мониторинга на уровне кортежей и обновляется, когда в течение окна мониторинга замечен доступ. В конце периода наблюдения строится список k самых востребованных кортежей. Программа мониторинга собирает эти списки и порождает глобальный список k самых горячих кортежей – это и есть данные, подлежащие миграции. В качестве побочного эффекта определяются «холодные» кортежи; особый интерес представляют кортежи, которые раньше были горячими, а затем стали холодными. Продолжительность окна мониторинга и величина k – параметры, задаваемые администратором системы.

2.4.3. Инкрементная реконфигурация

Как уже было сказано, наивный подход к перепроектированию – заново проделать всю работу по секционированию и размещению. Хотя в средах,

где рабочая нагрузка меняется нечасто, это может представлять интерес, в большинстве случаев накладные расходы слишком велики. Лучше применять изменения инкрементно, осуществляя миграцию данных; иными словами, мы находим, на какие данные повлияло изменение рабочей нагрузки, и перемещаем их с одного узла на другой[1]. Так что в этом разделе мы будем заниматься инкрементными подходами.

Базируясь на идеях из предыдущего раздела, можно предложить очевидный подход – использовать алгоритм инкрементного разрезания графа, который реагировал бы на изменения в описанном выше представлении графа. Так сделано в системах SWORD и AdaptCache, в которых использование представлено гиперграфами и производится их инкрементное разрезание. Сам факт инкрементного разрезания инициирует миграцию данных с целью реконфигурации.

В вышеупомянутой системе E-Store принят более изощренный подход. После того как множество горячих кортежей построено, подготавливается план миграции, который показывает, какие горячие кортежи следует переместить и как необходимо переразместить холодные кортежи. Эту задачу можно поставить как задачу оптимизации, в которой требуется создать сбалансированную нагрузку на вычислительные узлы (баланс определяется как средняя нагрузка по всем узлам ± пороговая величина), но решить эту задачу оптимизации в режиме реального времени и тем обеспечить онлайновую реконфигурацию нелегко, поэтому для генерации плана реконфигурации применяются приближенные методы (например, жадные алгоритмы или первый подходящий). По существу, сначала определяются подходящие узлы для каждого горячего кортежа, затем, если это необходимо для устранения оставшегося дисбаланса, обрабатываются холодные кортежи, которые перемещаются блоками. То есть в плане реконфигурации миграция горячих кортежей реализуется на индивидуальной основе, а холодных – на групповой. Частью плана является координирующий узел, который будет управлять миграцией, а весь план подается на вход системы реконфигурации Squall.

Squall производит реконфигурацию и миграцию данных за три шага. На первом шаге указанный в плане координатор инициализирует систему, готовя ее к миграции. Для этого нужно с помощью транзакционного механизма получить монопольный доступ ко всем секциям (см. главу 5). Затем координатор просит каждый узел идентифицировать кортежи, которые будут вынесены из локальной секции, и кортежи, которые будут внесены в нее. Этот анализ производится с привлечением метаданных, поэтому завершается быстро, после чего каждый узел уведомляет координатора, и инициализирующая транзакция завершается. На втором шаге координатор поручает каждому узлу выполнить миграцию данных. Это самый сложный этап, потому что к перемещаемым данным могут одновременно предъявляться запросы. Если запрос выполняется в узле, куда данные должны быть перенесены согласно плану, но еще не попали, то Squall «вытягивает» недостаю-

[1] Исследования в этой области посвящены исключительно горизонтальному секционированию, и мы тоже будем рассматривать только этот случай. Это значит, что единицей миграции является кортеж.

щие кортежи, чтобы все-таки обработать запрос. Это делается в дополнение к нормальной миграции данных по плану реконфигурации. Иными словами, чтобы вовремя выполнить запросы, Squall осуществляет перемещение по требованию, помимо нормальной миграции. После завершения этого шага каждый узел информирует координатора, который затем приступает к последнему шагу и уведомляет каждый узел о том, что реконфигурация закончена. Эти три шага необходимы, чтобы Squall могла выполнить миграцию, не останавливая обработку пользовательских запросов.

Другой подход называется *крекингом базы данных*; это адаптивная техника индексирования, ориентированная на динамичную, трудно предсказуемую рабочую нагрузку и такие сценарии, где мало или совсем нет свободного времени, которое можно было бы уделить анализу рабочей нагрузки и построению индексов. Метод крекинга постоянно реорганизует данные, чтобы они лучше соответствовали предъявляемым запросам. Каждый запрос рассматривается как рекомендация о том, как хранить данные. Процедура крекинга учитывает их, выполняя частичное и инкрементное построение и уточнение индексов как часть обработки запроса. Реагируя на каждый запрос простыми действиями, крекинг базы данных мгновенно адаптируется к изменяющейся рабочей нагрузке. По мере поступления запросов индексы становятся все точнее, а производительность улучшается и в конечном итоге достигает оптимального уровня, который мы получили бы, настраивая систему вручную.

Основная идея оригинального подхода к крекингу базы данных состояла в том, чтобы система реорганизовывала по одному столбцу данных за раз и только тогда, когда к нему обращается какой-то запрос. Иными словами, при реорганизации используется тот факт, что данные уже прочитаны, и принимается решение, как улучшить их организацию. По сути дела, оригинальный подход перегружает оператор выборки из базы данных и использует заданные в запросе предикаты, чтобы определить, как реорганизовать соответствующий столбец. Когда атрибут *A* впервые затребуется запросом, создается копия исходного столбца *A*, называемая крекинговым столбцом *A*. Каждый оператор выборки *A* инициирует физическую реорганизацию крекингового столбца, основанную на указанном в запросе диапазоне. Записи, в которых ключ меньше нижней границы, перемещаются под нижнюю границу, а записи, в которых ключ больше верхней границы, перемещаются за верхнюю границу. Информация о секционировании для каждого крекингового столбца хранится в AVL-дереве, крекинговом индексе. Будущие запросы к столбцу *A* ищут в крекинговом индексе секцию, в которую попадает запрошенный диапазон. Если запрошенный ключ уже присутствует в индексе, т. е. если прошлые запросы производили крекинг точно по такому диапазону, то оператор выборки может вернуть результат немедленно. В противном случае оператор выборки динамически уточняет столбец, т. е. реорганизуются только секции (участки) столбца с данными, удовлетворяющими предикатам (не более двух секций на границах диапазона). Постепенно столбец становится все более «упорядоченным», количество участков нем увеличивается, но сами они уменьшаются.

Идею крекинга базы данных и основы соответствующей техники можно обобщить, применив к секционированию данных в распределенной СУБД,

т. е. размещать данные в узлах, используя поступающие запросы как рекомендации. Всякий раз, как узлу нужны некоторые данные для локального запроса, а этих данных не оказывается, система может считать это указанием на необходимость переместить данные в этот узел. Но, в отличие от методов крекинга базы данных, размещенной в памяти, когда система немедленно реагирует на каждый запрос, в распределенной СУБД перемещение данных обходится дороже. Однако по той же причине выигрыш, который получат будущие запросы, более весомый. Этот компромисс уже изучался при применении различных вариаций крекинга к оптимизации данных на диске. Итог двоякий: (1) вместо того чтобы реагировать на каждый запрос, следует дождаться, когда данные о рабочей нагрузке станут более полными, и только тогда приступать к дорогостоящей реорганизации данных; (2) нужно применять «более тяжеловесные» виды реорганизации, поскольку чтение и запись данных обходятся дороже, чем при операциях в памяти. Мы ожидаем, что в будущем применение таких адаптивных методов индексирования получит развитие в ситуациях, когда предсказать рабочую нагрузку нелегко и не хватает времени для полной сортировки и секционирования данных до поступления первого запроса.

2.5. Каталог данных

И напоследок мы обсудим вопрос о каталоге данных. Система должна хранить и поддерживать схему распределенной базы данных. Эта информация необходима на этапе оптимизации распределенного запроса, как мы покажем ниже. Информация о схеме хранится в *каталоге данных*, представляющем собой метабазу, содержащую определения схемы и отображений, статистику использования, сведения о контроле доступа и т. п.

В случае распределенной СУБД схема определяется как на глобальном уровне (глобальная концептуальная схема – ГКС), так и в локальных узлах (локальные концептуальные схемы – ЛКС). ГКС определяет базу данных в целом, а ЛКС описывает данные в конкретном узле. Следовательно, существует два типа каталогов: *глобальный каталог* (ГК), который описывает базу данных так, как видит ее пользователь, и локальный каталог, описывающий локальные отображения и схему в каждом узле. Таким образом, компоненты управления локальной базой данных интегрированы с помощью функций глобальной СУБД.

Как уже было сказано, каталог сам является базой данных, содержащей *метаданные* о собственно данных, хранящихся в базе. Поэтому методы, рассмотренные в этой главе применительно к проектированию распределенной базы данных, применимы и к управлению каталогом, только гораздо проще. Коротко говоря, каталог может быть либо *глобальным* для всей базы данных, либо *локальным* для каждого узла. Иными словами, может существовать либо один каталог, содержащий информацию обо всех данных в базе, либо несколько каталогов, каждый из которых содержит информацию о данных, хранящихся в одном узле. Во втором случае мы можем либо построить иерархию

каталогов для упрощения поиска, либо реализовать стратегию распределенного поиска, подразумевающую значительный сетевой обмен информацией между узлами, где находятся каталоги.

Вторая проблема – репликация. Может существовать *единственная* копия каталога или *несколько* копий. Наличие нескольких копий повышает надежность, потому что вероятность добраться хотя бы до одной работоспособной копии больше. Кроме того, уменьшаются задержки при доступе к каталогу, поскольку меньше конкуренция и копии каталога можно найти поблизости. С другой стороны, поддержание каталога в актуальном состоянии становится гораздо более трудной задачей, потому что необходимо обновлять несколько копий. Поэтому выбор должен зависеть от среды, в которой работает система, и учитывать такие факторы, как требования ко времени отклика, размер каталога, вычислительные возможности компьютеров в узлах, надежность и изменчивость каталога (т. е. количество изменений в базе данных, требующих изменения каталога).

2.6. Заключение

В этой главе мы описали методы, используемые при проектировании распределенной базы данных, уделив особое внимание вопросам секционирования и размещения. Мы подробно обсудили алгоритмы фрагментации реляционной схемы разными способами. Эти алгоритмы разрабатывались независимо, и не существует базовой методологии проектирования, объединяющей методы горизонтального и вертикального секционирования. Начав с глобального отношения, можно применить алгоритмы для его разложения на несколько фрагментов как по вертикали, так и по горизонтали. Однако нет алгоритмов, которые разлагают глобальное отношение на фрагменты, часть из которых секционирована по горизонтали, а часть по вертикали. Часто отмечают, что в большинстве реальных ситуаций фрагментация смешанная, т. е. отношение разлагается как на вертикальные, так и на горизонтальные фрагменты, но исследований по общей методологии такого рода не ведется. Если двигаться в этом направлении, то нужна методология проектирования распределения, которая включает алгоритмы горизонтальной и вертикальной фрагментации как часть более общей стратегии. Такая методология должна была бы принимать глобальное отношение и набор критериев проектирования и выдавать множество фрагментов, одни из которых получены в результате горизонтальной, а другие в результате вертикальной фрагментации.

Мы также обсудили методы, которые не разделяют шаги фрагментации и размещения, – способ фрагментации диктует, как производить размещение, или наоборот. У этих методов есть две характерные особенности. Во-первых, все они относятся к горизонтальной фрагментации. Во-вторых, они более детальны, единицей размещения является кортеж; фрагменты в каждом узле «возникают» как объединение кортежей одного отношения, назначенных этому узлу.

Наконец, мы обсудили адаптивные методы, учитывающие изменение рабочей нагрузки. Как правило, они тоже ориентированы на горизонтальную фрагментацию, но ведут мониторинг изменений рабочей нагрузки (в терминах множества запросов и паттернов доступа) и соответственно подстраивают фрагментацию данных. Наивный способ реализовать эту идею – запускать алгоритм секционирования пакетно, но, по очевидным причинам, это нежелательно. Поэтому лучшие алгоритмы из этого класса модифицируют распределение данных инкрементно.

2.7. Библиографические замечания

Проектирование распределенных баз данных систематически изучалось с момента становления этой технологии. Пространство проектирования описано в ранней работе [Levin and Morgan 1975]. Работы [Davenport 1981], [Ceri et al. 1983] и [Ceri et al. 1987] содержат хорошие обзоры методологии проектирования. В работе [Ceri and Pernici 1985] обсуждается конкретная методология DATAID-D, похожая на ту, что изображена на рис. 2.1. Другие попытки разработать методологию имеются в статьях [Fisher et al. 1980], [Dawson 1980], [Hevner and Schneider 1980] и [Mohan 1979].

В этой главе рассмотрено большинство известных результатов по фрагментации. Работы по фрагментации в распределенных базах данных поначалу были посвящены только горизонтальной фрагментации. Обсуждение этой темы основано главным образом на работах [Ceri et al. 1982b] и [Ceri et al. 1983]. Секционирование данных в параллельных СУБД рассматривается в работе [DeWitt and Gray 1992]. Вопрос о вертикальной фрагментации при проектировании распределения изучался в нескольких работах (см., например, [Navathe et al. 1984] и [Sacca and Wiederhold 1985]). Начало исследований по вертикальной фрагментации восходит к диссертации Хоффера [Hoffer 1975, Hoffer and Severance 1975], а также к работам [Niamir 1978] и [Hammer and Niamir 1979]. В работе [McCormick et al. 1972] представлен алгоритм энергии связей, который был адаптирован к вертикальной фрагментации в работах [Hoffer and Severance 1975] и [Navathe et al. 1984].

Исследования в области размещения файлов в глобальных сетях начались с работ Чу [Chu 1969, 1973]. Обзор ранних работ имеется в отличной статье [Dowdy and Foster 1982]. Некоторые теоретические результаты приводятся в статьях [Grapa and Belford 1977] и [Kollias and Hatzopoulos 1981]. Работы по размещению данных восходят к середине 1970-х годов ([Eswaran 1974] и другие). В своей ранней работе Левин и Морган [Levin and Morgan 1975] сосредоточились на размещении данных, но впоследствии стали рассматривать совместное размещение данных и программ [Morgan and Levin 1977]. Задача о распределенном размещении данных изучалась в разных специализированных постановках. Были проведены исследования по размещению компьютеров и данных в глобальной сети [Gavish and Pirkul 1986]. Пропускная способность каналов в контексте размещения данных исследовалась в работе [Mahmoud and Riordon 1976], размещение данных в суперкомпьютерных си-

стемах – в работе [Irani and Khabbaz 1982], а в кластерах процессоров – в работе [Sacca and Wiederhold 1985]. Интересна работа [Apers 1981], в которой решается задача об оптимальном размещении узлов виртуальной сети, а затем ищется наилучшее соответствие между узлами виртуальной и физической сетей. Изоморфизм задачи размещения данных и задачи размещения складов одного товара установлен в работе [Ramamoorthy and Wah 1983]. Другие подходы изучались в следующих работах: задача о рюкзаке [Ceri et al. 1982a], метод ветвей и границ [Fisher and Hochbaum 1980], алгоритмы потоков в сетях [Chang and Liu 1982].

Подход к комбинированному секционированию в системе Schism (раздел 2.3.2) описан в работе [Curino et al. 2010], а система SWORD – в работе [Quamar et al. 2013]. Другие работы в том же ключе – [Zilio 1998], [Rao et al. 2002] и [Agrawal et al. 2004] – в основном посвящены параллельным СУБД.

Первые адаптивные подходы обсуждаются в работе [Wilson and Navathe 1986]. Ограниченное перепроектирование, в частности вопрос о материализации, изучается в работах [Rivera-Vega et al. 1990, Varadarajan et al. 1989]. Полное перепроектирование и материализация изучались в работах [Karlapalem et al. 1996, Karlapalem and Navathe 1994, Kazerouni and Karlapalem 1997]. В работе [Kazerouni and Karlapalem 1997] описана пошаговая методология перепроектирования, которую мы рассматривали в разделе 2.4. Система AdaptCache описана в работе [Asad and Kemme 2016].

Влияние изменений рабочей нагрузки на распределенные и параллельные СУБД, а также желательность локализации данных для каждой транзакции изучались в работах [Pavlo et al. 2012] и [Lin et al. 2016]. Имеется ряд работ, в которых рассматривается адаптивное секционирование при наличии таких изменений. Мы в своем обсуждении ограничились системой E-Store [Taft et al. 2014]. В E-Store с целью создания плана миграции реализованы подсистемы E-Monitor для мониторинга и обнаружения изменений рабочей нагрузки и E-Planner для нахождения элементов, на которые эти изменения повлияли. Для миграции как таковой применяется оптимизированная версия системы Squall [Elmore et al. 2015]. Есть и другие работы на ту же тему, например система P-Store [Taft et al. 2018] предсказывает нагрузку (в отличие от E-Store, которая реагирует на ее изменения).

Выявление изменений рабочей нагрузки на основе анализа журналов изучалось в работе [Levandoski et al. 2013].

Одна из работ, посвященных обнаружению изменений рабочей нагрузки при автономных вычислениях, – [Holze and Ritter 2008]. Система Apollo, которую мы упоминали при обсуждении вопроса о том, как находить затронутые изменением рабочей нагрузки данные, и которая обобщает отдельные запросы, создавая шаблоны запросов, с целью предсказательных вычислений, описана в работе [Glasbergen et al. 2018].

Концепция крекинга базы данных в контексте столбцовых хранилищ в основной памяти изучалась в работах [Idreos et al. 2007b, Schuhknecht et al. 2013]. Алгоритмы крекинга были адаптированы для решения многих основополагающих вопросов архитектуры базы данных, например: обновление с целью инкрементного и адаптивного учета изменений данных [Idreos et al. 2007a], анализ запросов с несколькими атрибутами для реорганизации отношений

целиком, а не отдельных столбцов [Idreos et al. 2009], использование оператора соединения как триггера адаптации [Idreos 2010], управление конкурентностью для решения проблемы, вследствие которой крекинг по существу преобразует операции чтения в операции записи [Graefe et al. 2014, 2012], и логика секционирования со слиянием для создания алгоритмов крекинга, способных находить компромисс между сходимостью индекса и стоимостью инициализации [Idreos et al. 2011]. Кроме того, были разработаны специальные стресс-тесты для проверки различных критических характеристик, в т. ч. скорости адаптации алгоритма [Graefe et al. 2010]. Стохастический крекинг базы данных [Halim et al. 2012] – идея о том, как добиться робастности при различных рабочих нагрузках, а в работе [Graefe and Kuno 2010b] показано, как можно применить адаптивное индексирование к ключевым столбцам. Наконец, в недавних работах по параллельному адаптивному индексированию изучаются эффективные с точки зрения использования ЦП реализации и предлагаются алгоритмы крекинга, извлекающие выгоду из наличия нескольких ядер [Pirk et al. 2014, Alvarez et al. 2014] и даже из времени, когда процессор простаивает [Petraki et al. 2015].

Концепция крекинга базы данных была также распространена на более общие решения о структуре хранения, а именно на реорганизацию способа хранения данных (по столбцам или по строкам) в соответствии с поступающими запросами [Alagiannis et al. 2014]. Она даже применялась к решению вопроса о том, какие данные следует загружать [Idreos et al. 2011, Alagiannis et al. 2012]. Крекинг изучался и в контексте Hadoop [Richter et al. 2013] для локального индексирования в каждом узле и для улучшения более традиционного дискового индексирования, когда данные читаются страницами, а запись реорганизованных данных назад вынужденно рассматривается как значительные накладные расходы [Graefe and Kuno 2010a].

Упражнения

Задача 2.1 (*). Рассмотрим отношение EMP на рис. 2.2, и пусть p_1: TITLE < "Programmer" и p_2: TITLE > "Programmer" – два простых предиката. Предположим, что на строках символов определен алфавитный порядок.

- **a)** Выполните горизонтальную фрагментацию EMP относительно $\{p_1, p_2\}$.
- **b)** Объясните, почему получившаяся фрагментация (EMP_1, EMP_2) не удовлетворяет правилам корректности.
- **c)** Модифицируйте предикаты p_1 и p_2, так чтобы при фрагментации EMP соблюдались правила корректности. Для этого измените предикаты, составьте все элементарные конъюнкции, выведите соответствующие импликации и выполните горизонтальную фрагментацию EMP на основе этих элементарных конъюнкций. После этого покажите, что результат обладает свойствами полноты, реконструируемости и дизъюнктности.

Задача 2.2 (*). Рассмотрим отношение ASG на рис. 2.2. Пусть имеется два приложения, обращающихся к ASG. Первое выполняется в пяти узлах и пытается найти продолжительность назначения на проекты работников с заданными

номерами. Предположим, что данные о менеджерах, консультантах, инженерах и программистах хранятся в четырех разных узлах. Второе приложение выполняется в двух узлах, причем работники со сроком назначения меньше 20 месяцев хранятся в одном узле, а с большим сроком – в другом. Спроектируйте главную горизонтальную фрагментацию ASG на основе этой информации.

Задача 2.3. Рассмотрим отношения EMP и PAY на рис. 2.2. Они горизонтально фрагментированы следующим образом:

$$\mathrm{EMP}_1 = \sigma_{\mathrm{TITLE}=\text{"Elect. Eng."}}(\mathrm{EMP})$$
$$\mathrm{EMP}_2 = \sigma_{\mathrm{TITLE}=\text{"Syst. Anal."}}(\mathrm{EMP})$$
$$\mathrm{EMP}_3 = \sigma_{\mathrm{TITLE}=\text{"Mech. Eng."}}(\mathrm{EMP})$$
$$\mathrm{EMP}_4 = \sigma_{\mathrm{TITLE}=\text{"Programmer"}}(\mathrm{EMP})$$
$$\mathrm{PAY}_1 = \sigma_{\mathrm{SAL}\geq 30000}(\mathrm{PAY})$$
$$\mathrm{PAY}_2 = \sigma_{\mathrm{SAL}<30000}(\mathrm{PAY})$$

Нарисуйте граф соединений $\mathrm{EMP} \ltimes_{\mathrm{TITLE}} \mathrm{PAY}$. Является ли граф простым или разрезанным? Если он разрезан, то модифицируйте фрагментацию EMP или PAY, так чтобы этот граф стал простым.

Задача 2.4. Приведите пример матрицы *CA*, для которой точка разделения не единственная, а секционирование находится в середине матрицы. Вычислите, сколько операций сдвига необходимо, чтобы получить единственную однозначно определенную точку разделения.

Задача 2.5 (**). Рассмотрим отношение PAY на рис. 2.2, пусть p_1 : SAL < 30 000 и p_2 : SAL ⩾ 30 000 – два простых предиката. Выполните горизонтальную фрагментацию PAY относительно этих предикатов и получите PAY_1 и PAY_2. Пользуясь этой фрагментацией PAY, произведите дополнительную производную горизонтальную фрагментацию EMP. Докажите, что фрагментация EMP полна, допускает реконструкцию и дизъюнктна.

Задача 2.6 (**). Пусть $Q = \{q_1, \ldots, q_5\}$ – множество запросов, $\mathrm{A} = \{\mathrm{A}_1, \ldots, \mathrm{A}_5\}$ – множество атрибутов, $S = \{S_1, S_2, S_3\}$ – множество узлов. Матрица на рис. 2.25a описывает показатели использования атрибутов, а матрица на рис. 2.25b – частоты доступа приложений. Предположим, что $ref_i(q_k) = 1$ для всех q_k и S_i и что A_1 – ключевой атрибут. Воспользуйтесь алгоритмами энергии связей и вертикального секционирования, чтобы получить вертикальную фрагментацию множества атрибутов A.

	A_1	A_2	A_3	A_4	A_5
q_1	0	1	1	0	1
q_2	1	1	1	0	1
q_3	1	0	0	1	1
q_4	0	0	1	0	0
q_5	1	1	1	0	0

(a)

	S_1	S_2	S_3
q_1	10	20	0
q_2	5	0	10
q_3	0	35	5
q_4	0	10	0
q_5	0	15	0

(b)

Рис. 2.25 ❖ Показатели использования атрибутов и частоты доступа приложений в упражнении 3.6

Задача 2.7 (**). Напишите алгоритм производной горизонтальной фрагментации.

Задача 2.8 (**). Пусть к представлению

```
CREATE VIEW   EMPVIEW(ENO, ENAME, PNO, RESP)
AS    SELECT  EMP.ENO, EMP.ENAME, ASG.PNO, ASG.RESP
      FROM    EMP JOIN ASG
      WHERE   DUR=24
```

производится обращение со стороны приложения q_1, выполняемого в узлах 1 и 2, с частотами 10 и 20 соответственно. Пусть также имеется следующий запрос q_2

```
SELECT ENO, DUR
FROM   ASG
```

который выполняется в узлах 2 и 3 с частотами 20 и 10 соответственно. На основе этой информации постройте матрицу $use(q_i, A_j)$ для атрибутов обоих отношений EMP и ASG. Также постройте матрицу родства, содержащую все атрибуты EMP и ASG. Наконец, преобразуйте матрицу родства, так чтобы ее можно было использовать для расщепления отношения на два вертикальных фрагмента с помощью эвристики или алгоритма энергии связей BEA.

Задача 2.9 (**). Формально определите все три критерия корректности для производной горизонтальной фрагментации.

Задача 2.10 (*). Пусть дано отношение $R(K, A, B, C)$ (где K – ключ) и запрос

```
SELECT *
FROM   R
BEAR.A=10 AND R.B=15
```

- **a)** Какой результат получится при выполнении главной горизонтальной фрагментации по этому запросу?
- **b)** Порождает ли в этом случае алгоритм COM_MIN полный и минимальный набор предикатов? Обоснуйте свой ответ.

Задача 2.11 (*). Покажите, что алгоритм энергии связей порождает одинаковые результаты вне зависимости от того, применяется он к строкам или к столбцам.

Задача 2.12 (**). Модифицируйте алгоритм SPLIT, так чтобы он допускал n-путевое секционирование, и оцените сложность получившегося алгоритма.

Задача 2.13 (**). Формально определите все три критерия корректности для гибридной фрагментации.

Задача 2.14. Как порядок применения двух базовых схем фрагментации в гибридной фрагментации влияет на конечный результат?

Задача 2.15 (**). Опишите, как правильно смоделировать следующие аспекты задачи размещения базы данных:

- **a)** связи между фрагментами;

b) обработка запросов;
c) гарантии целостности;
d) механизмы управления конкурентностью.

Задача 2.16 (**). Рассмотрите различные эвристические алгоритмы для задачи размещения базы данных.

a) Придумайте и обсудите некоторые разумные критерии, по которым можно сравнивать эти эвристики.
b) Сравните эвристические алгоритмы по этим критериям.

Задача 2.17 (*). Выберите один из эвристических алгоритмов решения задачи размещения базы данных и напишите реализующую его программу.

Задача 2.18 (**). Рассмотрим те же условия, что в упражнении 3.8. Предположим еще, что в 60 % случаев запрос q_1 обновляет атрибуты PNO и RESP представления EMPVIEW, а атрибут ASG.DUR никогда не обновляется через EMPVIEW. Будем считать, что скорость передачи данных между узлами 1 и 2 в два раза меньше, чем между узлами 2 и 3. При таких условиях найдите разумную фрагментацию ASG и EMP и оптимальную репликацию и размещение фрагментов в предположении, что стоимость хранения не играет роли, но копии поддерживаются в согласованном состоянии.

Указание. Рассмотрите горизонтальную фрагментацию ASG на основе предиката DUR = 24 и соответствующую производную горизонтальную фрагментацию EMP. Также возьмите матрицу родства, полученную в примере 2.7 для EMP и ASG вместе, и посмотрите, имеет ли смысл производить вертикальную фрагментацию ASG.

Глава 3

Контроль распределенных данных

Важное требование к СУБД – поддержка контроля данных, т. е. способов доступа к данным из программы на языке высокого уровня. Обычно контроль данных включает управление представлениями, контроль доступа и контроль семантической целостности. Неформально говоря, эти функции должны гарантировать, что *авторизованные* пользователи выполняют *корректные* операции в базе данных, обеспечивая тем самым поддержание целостности данных. Вопрос о том, какие функции необходимы для поддержания физической целостности при наличии конкурентных операций доступа и возможности отказов, изучается отдельно в главе 5 в контексте управления транзакциями. В реляционных СУБД к контролю данных можно подходить единообразно. Представления, авторизацию и семантические ограничения целостности можно определить как правила, за соблюдением которых система следит автоматически. Нарушение любого правила в процессе выполнения операции обычно приводит к отказу в некоторых последствиях операции (например, откату обновлений) или распространению последствий (например, обновлению связанных данных) для сохранения целостности.

Определение таких правил – часть администрирования базы данных, работы, которая обычно возлагается на администратора базы данных (АБД, англ. DBA). Это лицо также отвечает за применение политик организации. Для централизованных СУБД имеются хорошо известные решения по управлению данными. В этой главе мы обсудим, как эти решения можно обобщить на распределенные СУБД. Стоимость принудительного контроля данных в терминах потребления ресурсов высока даже в централизованных СУБД и может оказаться неприемлемой в распределенной среде.

Поскольку правила контроля данных нужно хранить, управление распределенным каталогом имеет отношение и к материалу этой главы. Каталог распределенной СУБД можно рассматривать как распределенную базу данных. Существует несколько способов хранения определений правил, зависящих от организации каталога. Информацию каталога можно хранить поразному в зависимости от типа; иными словами, часть информации может полностью реплицироваться, а другая часть – оставаться распределенной. Например, реплицироваться может информация, полезная на этапе компиляции, скажем информация о контроле доступе. В этой главе мы будем

постоянно подчеркивать влияние способа управления каталогом на производительность механизмов контроля данных.

Эта глава организована следующим образом: управление представлениями – тема раздела 3.1, контроль доступа представлен в разделе 3.2, а контроль семантической целостности – в разделе 3.3. В каждом разделе мы сначала кратко описываем решение для централизованной СУБД, а затем приводим распределенное решение, которое зачастую оказывается обобщением централизованного, хотя и более трудным.

3.1. Управление представлениями

Одно из основных преимуществ реляционной модели заключается в полной логической независимости данных. Как было отмечено в главе 1, внешние схемы позволяют группам пользователей иметь собственное *представление* базы данных. В реляционной системе представление – это *виртуальное отношение*, определяемое как результат запроса к *базовым отношениям*, но, в отличие от базового, не материализованное в базе данных. Внешняя схема может быть определена как набор представлений и (или) базовых отношений. Помимо использования во внешних схемах, представления полезны для обеспечения безопасности данных очень простым способом. Выбирая только подмножество базы данных, представление *скрывает* некоторые данные. Если пользователи могут обращаться к базе только через представления, то они не могут ни видеть скрытые данные, ни манипулировать ими, а стало быть, эти данные оказываются в безопасности.

Далее в этом разделе мы рассмотрим управление представлениями в централизованной и распределенной системе, а также проблемы, возникающие при обновлении представлений. Заметим, что в распределенной СУБД представление может быть построено на базе распределенных отношений, и для доступа к нему потребуется выполнить соответствующий распределенный запрос. Важная проблема в распределенных СУБД – обеспечить эффективную материализацию представлений. Мы увидим, что концепция материализованных представлений помогает в решении данной, а равно других проблем, однако при этом необходимы эффективные методы обслуживания материализованных представлений.

3.1.1. Представления в централизованных СУБД

В большинстве реляционных СУБД используется механизм создания представлений на основе базовых отношений с помощью реляционного запроса (впервые это было предложено в проектах INGRES и System R). Для определения представления с запросом выборки нужно связать имя.

Пример 3.1. Представление, отбирающее только системных аналитиков (`SYS-AN`) из отношения `EMP`, можно определить с помощью такого SQL-запроса:

```
CREATE VIEW SYSAN(ENO, ENAME) AS
  SELECT ENO, ENAME
  FROM   EMP
  WHERE  TITLE = "Syst. Anal."                                    ♦
```

Единственный результат этой команды – сохранение определения представления в каталоге. Больше никакой информации запоминать не нужно. Когда представление создается, результат определяющего его запроса (т. е. отношение, содержащее атрибуты ENO и ENAME для всех системных аналитиков, – рис. 3.1) *не* порождается. Но с представлением SYSAN можно обращаться, как с обычным базовым отношением.

SYSAN

ENO	ENAME
E2	M. Smith
E5	B. Casey
E8	J. Jones

Рис. 3.1 ❖ Отношение, соответствующее представлению SYSAN

Пример 3.2. Запрос «Найти имена всех системных аналитиков, а также номера проектов, в которых они заняты, и их роль в проекте», который обращен к представлению SYSAN и отношению ASG, можно записать в виде

```
SELECT ENAME, PNO, RESP
FROM   SYSAN NATURAL JOIN ASG                                     ♦
```

Для преобразования запроса, выраженного в терминах представлений, в запрос в терминах базовых отношений необходимо выполнить *модификацию запроса*. Это означает, что переменные определяются над базовыми отношениями, а формулировка запроса объединяется (посредством AND) с определением представления.

Пример 3.3. Предыдущий запрос можно преобразовать в такой:

```
SELECT ENAME, PNO, RESP
FROM   EMP NATURAL JOIN ASG
WHERE  TITLE = "Syst. Anal."
```

Результат этого запроса показан на рис. 3.2. ♦

ENAME	PNO	TITLE
M. Smith	P1	Analyst
M. Smith	P2	Analyst
B. Casey	P3	Manager
J. Jones	P4	Manager

Рис. 3.2 ❖ Результат запроса, включающего представление SYSAN

Модифицированный запрос выражен в терминах базовых отношений, поэтому может быть обработан процессором запросом. Важно отметить, что эту обработку можно выполнить на этапе компиляции. Механизм представлений можно также использовать для уточнения контроля доступа, включив в запрос подмножество объектов. Ключевое слово USER обозначает идентификатор текущего пользователя, с его помощью можно ограничить набор данных, видимых пользователю.

Пример 3.4. Представление ESAME разрешает любому пользователю доступ только к работникам, занимающим ту же должность, что он сам:

```
CREATE VIEW ESAME AS
  SELECT *
  FROM   EMP E1, EMP E2
  WHERE  E1.ENO = E2.ENO
  AND    E1.ENO = USER
```

В этом определении * означает «все атрибуты», а две кортежные переменные (E1 и E2) над отношением EMP необходимы, чтобы выразить соединение одного кортежа EMP (соответствующего текущему пользователю) со всеми кортежами EMP, описывающими работников в той же должности. Например, следующий запрос, выполненный пользователем J. Doe:

```
SELECT *
FROM   ESAME
```

– возвращает отношение, показанное на рис. 3.3. Отметим, что сам пользователь J. Doe также входит в результат. Если пользователь, создавший представление ESAME, является инженером-электротехником, то это представление описывает множество всех электротехников. ♦

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E2	L. Chu	Elect. Eng.

Рис. 3.3 ❖ Результат запроса к представлению ESAME

Для определения представлений можно использовать сколь угодно сложные запросы, включающие выборку, проекцию, соединение, агрегатные функции и т. д. Все представления опрашиваются как базовые отношения, но не каждое представление можно обновлять, как базовое отношение. Обновление через представление может быть обработано автоматически, только если его можно корректно распространить на базовые отношения. Все представления можно разделить на две группы: обновляемые и необновляемые. Представление является обновляемым, только если его обновление допускает однозначное распространение на базовые отношения. Так, представление SYSAN выше обновляемое; например, вставка нового системного аналитика (201, Smith) преобразуется во вставку нового работника (201, Smith, Syst.

Anal.). Если бы все атрибуты, кроме TITLE, были скрыты представлением, то им были бы присвоены *значения null*.

Пример 3.5. Однако следующее представление, в котором используется естественное соединение (т. е. эквисоединение двух отношений по общему атрибуту), не является обновляемым:

```
CREATE VIEW EG(ENAME, RESP) AS
SELECT DISTINCT ENAME, RESP
FROM   EMP NATURAL JOIN ASG
```

Например, удаление кортежа (Smith, Analyst) невозможно распространить однозначным образом. Имеет смысл как операция удаления работника Smith из отношения EMP, так и операция удаления аналитика из отношения ASG, но система не знает, какую из них выбрать. ♦

Современные системы имеют много ограничений в части поддержки обновлений через представления. Представление можно обновлять, только если оно определено над одним отношением и включает лишь операции выборки и проекции. Это оставляет за скобками представления, в определении которых участвуют соединения, агрегатные функции и т. д. Но теоретически есть возможность реализовать автоматическую поддержку обновлений для более широкого класса представлений. Интересно отметить, что представления на основе соединения обновляемые, если включают ключи базовых отношений.

3.1.2. Представления в распределенных СУБД

В распределенной и централизованной СУБД представления определяются похожим образом. Однако в распределенной системе представление может быть основано на фрагментированных отношениях, хранящихся в разных узлах. Когда определяется представление, его имя и запрос выборки сохраняются в каталоге.

Поскольку прикладные программы могут использовать представления как базовые отношения, их определения следует хранить в каталоге так же, как описания базовых отношений. В зависимости от степени автономности узла, предлагаемой системой, определения представлений могут централизованно храниться в одном узле либо дублироваться частично или полностью. В любом случае сведения об ассоциации имени представления с узлом, где хранится его определение, следует дублировать. Если определения представления нет в узле, инициировавшем запрос, то понадобится удаленный доступ к узлу, на котором оно находится.

Преобразование запроса, выраженного в терминах представлений, в запрос, выраженный в терминах базовых отношений (которые потенциально могут быть фрагментированы), можно осуществить путем модификации запроса – так же, как в централизованной СУБД. При таком подходе определение запроса ищется в каталоге распределенной базы данных, а затем объединяется с самим запросом, в результате чего получается запрос к базо-

вым отношениям. Этот модифицированный запрос является *распределенным* и обрабатывается процессором распределенных запросов (см. главу 4). Процессор запросов отображает распределенный запрос на запрос к физическим фрагментам.

В главе 2 мы рассматривали различные способы фрагментации базовых отношений. Определение фрагментации на самом деле очень похоже на определение специальных представлений. Таким образом, для управления представлениями и фрагментами можно использовать унифицированный механизм. И точно так же можно обрабатывать реплицированные данные. Ценность такого механизма в том, что он позволяет упростить администрирование распределенной базы данных. Объекты, которыми манипулирует администратор, можно организовать в виде дерева – в его листьях находятся фрагменты, на основе которых можно строить отношения и представления. Поэтому администратор может повысить локальность ссылок, установив взаимно однозначное соответствие между представлениями и фрагментами. Например, можно реализовать представление SYSAN из примера 3.1 с помощью фрагмента в заданном узле, при условии что большинство пользователей, обращающихся к SYSAN, находятся в этом узле.

Вычисление представлений, построенных над распределенными отношениями, может обойтись дорого. Вполне вероятно, что в данной организации есть много пользователей, обращающихся к одному и тому же представлению, так что его приходится заново вычислять для каждого пользователя. В разделе 3.1.1 мы видели, что для вычисления представления его определение объединяется с определением запроса. Альтернативой вычислению является хранение фактических версий представления в виде так называемых *материализованных представлений*. В материализованном представлении хранятся кортежи представления, взятые из отношения базы данных; над ними можно даже строить индексы. Поэтому доступ к материализованному представлению гораздо быстрее, чем вычисление представления, особенно в распределенной СУБД, где базовые отношения могут быть удаленными. Впервые появившиеся в 1980-х годах, материализованные представления завоевали популярность в контексте хранилищ данных, поскольку позволили ускорить работу приложений оперативной аналитической обработки данных (OLAP). В хранилищах данных материализованные представления обычно включают агрегат (например, **SUM** или **COUNT**) и оператор группировки (**GROUP BY**), поскольку дают компактную сводку базы данных. Сегодня все основные СУБД поддерживают материализованные представления.

Пример 3.6. Следующее представление над отношением PROJ(PNO, PNAME, BUDGET, LOC) для каждого местоположения возвращает количество проектов и суммарный бюджет.

```
CREATE VIEW PL(LOC, NBPROJ, TBUDGET) AS
  SELECT LOC, COUNT(*),SUM(BUDGET)
  FROM   PROJ
  GROUP BY LOC
```

♦

3.1.3. Обслуживание материализованных представлений

Материализованное представление – это копия некоторых базовых данных, поэтому должно быть согласовано с этими данными, которые могут обновляться. *Обслуживанием представления* называется процесс его обновления с целью отразить изменение базовых данных. С материализацией представления связаны примерно такие же вопросы, как с репликацией базы данных (см. главу 6). Но главное различие заключается в том, что определения материализованных представлений, в особенности используемых при организации хранилищ данных, как правило, гораздо сложнее, чем определения реплик, и могут включать соединения, группировку и агрегирование. Еще одно важное различие – то, что репликация базы данных связана с более общей конфигурацией, например с наличием нескольких копий одних и тех же данных в разных узлах.

Политика обслуживания представлений позволяет АБД задавать, *когда* и *как* следует обновлять представление. Первый вопрос (когда обновлять) связан с согласованностью (представления и базовых данных) и эффективностью. Представление можно обновлять в одном из двух режимов: *немедленном* и *отложенном*. В немедленном режиме представление обновляется сразу же, как только транзакция обновляет базовые данные, используемые представлением. Если представление и базовые данные управляются разными СУБД, возможно в разных узлах, то для этого придется использовать распределенную транзакцию, например протокол двухфазной фиксации (см. главу 5). Главные достоинства немедленного обновления – то, что согласованность с базовыми данными поддерживается всегда и что запросы чтения выполняются быстро. Но за это приходится расплачиваться увеличенным временем выполнения транзакции, которая должна обновить и базовые данные, и надстроенные над ними представления. Кроме того, использовать распределенные транзакции не всегда легко.

На практике предпочтительнее отложенный режим, потому что представление обновляется в отдельной транзакции, не замедляя транзакции, обновляющие базовые данные. Транзакции обновления можно инициировать в разные моменты времени: *лениво*, т. е. непосредственно перед выполнением запроса к представлению; *периодически*, т. е. по заранее определенному расписанию, например один раз в день, или *принудительно*, т. е. после заранее определенного количества обновлений базовых данных. При ленивом обновлении запрос будет видеть последнее согласованное состояние базовых данных, но ценой увеличения времени, в которое включается обновление представления. При периодическом и принудительном обновлении запрос может видеть данные, не согласованные с последним состоянием базовых данных. Представления, для управления которыми применяется одна из этих двух стратегий, называются также *моментальными снимками*.

Второй вопрос (как обновлять представление) важен с точки зрения эффективности. Проще всего вычислить представление с нуля, воспользовавшись базовыми данными. Иногда эта стратегия самая эффективная, например

если изменилось большое подмножество базовых данных. Но часто бывает, что изменить нужно только небольшое подмножество данных представления. Тогда разумнее вычислять представление *инкрементно*, т. е. только изменения. Инкрементное обслуживание представления опирается на понятие дифференциального отношения. Пусть *u* – обновление отношения R. *Дифференциальными отношениями* R относительно *u* называются отношение R^+, содержащее все кортежи, вставленные в R в результате выполнения *u*, и R^-, содержащее все кортежи, удаленные из R в результате выполнения *u*. Если *u* – операция вставки, то R^- пусто. Если *u* – операция удаления, то R^+ пусто. Наконец, если *u* – модификация, то отношение R можно получить, вычислив $(V - V^-) \cup V^+$. Для вычисления изменений представления, т. е. V^+ и V^-, могут понадобиться не только дифференциальные, но и базовые отношения.

Пример 3.7. Рассмотрим представление EG из примера 3.5, построенное над базовыми отношениями EMP и ASG, и предположим, что его состояние выведено из данных в примере 3.1, так что EG содержит 9 кортежей (см. рис. 3.4). Пусть EMP+ включает кортеж (E9, B. Martin, Programmer), вставляемый в EMP, а ASG+ включает два кортежа (E4, P3, Programmer, 12) и (E9, P3, Programmer, 12), вставляемых в ASG. Изменение представления EG можно вычислить как

```
EG+ = (SELECT ENAME, RESP
       FROM   EMP NATURAL JOIN ASG+)
           UNION
      (SELECT ENAME, RESP
       FROM   EMP+ NATURAL JOIN ASG)
           UNION
      (SELECT ENAME, RESP
       FROM   EMP+ NATURAL JOIN ASG+)
```

что дает кортежи (B. Martin, Programmer) и (J. Miller, Programmer). Заметим, что здесь были бы полезны ограничения целостности, чтобы избежать бесполезной работы (см. раздел 3.3.2). В предположении, что отношения EMP и ASG связаны ограничением ссылочной целостности, согласно которому атрибут ENO в ASG должен присутствовать в EMP, вторая команда **SELECT** оказывается бесполезной, поскольку порождает пустое отношение. ♦

Для выполнения инкрементного обслуживания представлений придуманы эффективные методы, в которых используются как материализованные представления, так и базовые отношения. Они существенно различаются в части выразительности представлений, использования ограничений целостности и обработки вставок и удалений. По выразительности представления можно классифицировать как нерекурсивные, включающие внешние соединения и рекурсивные. Для нерекурсивных представлений, включающих только операции выборки, проекции и соединения (ВПС-представления), возможно, с исключением дубликатов, объединением и агрегированием, имеется элегантное решение – *алгоритм подсчета*. Одна из проблем вызвана тем, что один кортеж в представлении может быть выведен из нескольких кортежей в базовых отношениях, что затрудняет удаление из представления. Основная идея алгоритма подсчета состоит в том, чтобы для каждого кортежа

в представлении запоминать количество таких выведений и увеличивать (уменьшать) счетчики кортежей при вставке (удалении); если счетчик кортежа в представлении обратился в ноль, то этот кортеж можно удалять.

Пример 3.8. Рассмотрим представление EG на рис. 3.4. Каждый кортеж в EG выведен из одного (т. е. счетчик равен 1), за исключением кортежа (M. Smith, Analyst), который выведен из двух (счетчик равен 2). Теперь предположим, что кортежи (E2, P1, Analyst, 24) и (E3, P3, Consultant, 10) удалены из ASG. Тогда из EG следует удалить только кортеж (A. Lee, Consultant). ♦

ENAME	RESP
J. Doe	Manager
M. Smith	Analyst
A. Lee	Consultant
A. Lee	Engineer
J. Miller	Programmer
B. Casey	Manager
L. Chu	Manager
R. Davis	Engineer
J. Jones	Manager

Рис. 3.4 ❖ Состояние представления EG

Теперь опишем базовый алгоритм подсчета для обновления представления V, определенного над двумя отношениями R и S в виде запроса q(R, S). В предположении, что с каждым кортежем в V ассоциирован счетчик выведений, алгоритм состоит из трех шагов (см. алгоритм 3.1). Сначала применяется техника дифференциации, чтобы описать дифференциальные представления V^+ и V^- как запросы к представлению, базовым отношениям и дифференциальным отношениям. Затем вычисляются V^+ и V^- вместе с их счетчиками кортежей. И на третьем шаге изменения V^+ и V^- применяются к V путем прибавления положительных счетчиков, вычитания отрицательных и удаления кортежей, для которых счетчик стал равен нулю.

Алгоритм 3.1. COUNTING

Вход: V : представление, определенное как q(R, S); R, S: отношения; R^+, R^-: изменения R

begin

$V^+ = q^+(V, R^+, R, S)$

$V^- = q^-(V, R^-, R, S)$

вычислить V^+ с положительными счетчиками для вставленных кортежей

вычислить V^- с отрицательными счетчиками для удаленных кортежей

вычислить $(V - V^-) \cup V^+$, прибавив положительные счетчики и вычтя отрицательные

удалить из V все кортежи, для которых счетчик равен 0;

end

Алгоритм подсчета оптимален, потому что вычисляет только вставленные или удаленные кортежи представления. Однако ему необходим доступ к базовым отношениям. Поэтому базовые отношения должны храниться (быть может, в виде реплик) в узлах материализованного представления. Чтобы избежать доступа к базовым отношениям и, стало быть, получить возможность хранить представление в другом узле, для обслуживания представления должно быть достаточно его самого и дифференциальных отношений. Такие представления называются *самообслуживаемыми*.

Пример 3.9. Рассмотрим представление SYSAN из примера 3.1. Запишем его определение в виде SYSAN=q(EMP), понимая под этим, что представление определено запросом над q над EMP. Мы можем вычислить дифференциальные представления, используя только дифференциальные отношения, т. е. $\text{SYSAN}^+ = q(\text{EMP}^+)$ и $\text{SYSAN}^- = q(\text{EMP}^-)$. Таким образом, представление SYSAN самообслуживаемое. ♦

Самообслуживаемость зависит от выразительности представления, ее можно определить относительно типа обновления (вставка, удаление, модификация). Большинство ВПС-представлений не являются самообслуживаемыми относительно вставки, но часто являются самообслуживаемыми относительно удаления и модификации. Например, ВПС-представление самообслуживаемое относительно удаления из отношения R, если оно включает ключевые атрибуты R.

Пример 3.10. Рассмотрим представление EG из примера 3.5. Добавим в его определение атрибут ENO (являющийся ключом EMP). Это представление не самообслуживаемое относительно вставки. Например, после вставки кортежа в ASG необходимо выполнить соединение с EMP, чтобы получить атрибут ENAME для вставки в представление. Однако оно самообслуживаемое относительно удаления из EMP. Например, если из EMP удален один кортеж, то можно удалить из представления кортежи с тем же значением ENO. ♦

Мы обсудим две оптимизации, которые могут значительно уменьшить время обслуживания в алгоритме COUNTING. Первая оптимизация – материализовывать представления, соответствующие подзапросам входного запроса. Представление строится путем удаления некоторого подмножества отношений из запроса. Эти представления прогрессивно меньше и образуют иерархию. Такую иерархию, называемую деревом представления, строит метод F-IVM; в его корне находится входной запрос, в листьях – отношения, а представления во внутренних узлах определены запросами типа проекция–соединение–агрегат над дочерними узлами. Обновления отношения распространяются по дереву представления снизу вверх. Представления, находящиеся на пути от обновленного отношения к корню, обслуживаются методом обработки дельт из алгоритма COUNTING. Все остальные представления не изменяются; если они материализованы, то могут ускорить обработку дельт. Для ограниченного класса ациклических запросов, называемых q-иерархическими, такие деревья представлений допускают обновление любого входного отношения за постоянное время.

Во второй оптимизации используется асимметрия данных. Значения, встречающиеся в базе данных очень часто, считаются тяжелыми, осталь-

ные – легкими. В методе IVM$^{\varepsilon}$ применяются стратегии вычисления, чувствительные к асимметрии тяжелых и легких данных, в которых используются материализованные представления и вычисление дельт, как во всех вышеупомянутых алгоритмах обслуживания.

Мы приведем примеры этих двух оптимизаций для запроса, подсчитывающего количество треугольников в графе. Мы хотели бы обновлять счетчик треугольников немедленно и инкрементно после одного обновления графа, которое может сводиться к вставке или удалению ребра. Рассмотрим три копии `R`, `S`, `T` бинарного реберного отношения для графа с N ребрами. Мы запоминаем кратности кортежей во входных отношениях и представлениях (сколько раз был выведен каждый кортеж) в отдельном столбце `P`. В предположении, что схемы отношений имеют вид `(A,B,P_R)`, `(B,C,P_S)` и `(C,A,P_T)`, запрос для вычисления счетчика треугольников имеет вид:

```
CREATE VIEW Q(CNT) AS
  SELECT SUM(P_R * P_S * P_T) as CNT
  FROM   R NATURAL JOIN S NATURAL JOIN T
```

Вставка или удаление ребра инициирует обновление всех трех копий отношения. Мы обсудим обновление `R`, остальные два случая рассматриваются аналогично. Смоделируем это обновление в виде отношения `deltaR`, состоящего из одного кортежа (a, b, p), где (a, b) – обновленное ребро, а p – кратность. В соответствии с формализмом обобщенных отношений-мультимножеств мы моделируем вставки и удаления единообразно, считая, что кратности могут быть целыми числами – положительными и отрицательными. Тогда, чтобы вставить или удалить ребро три раза, мы просто увеличиваем или уменьшаем кратность p на 3.

Алгоритм COUNTING динамически вычисляет запрос `deltaQ`, который представляет изменение результата запроса; это тот же запрос, что и `Q`, но с заменой `R` на `deltaR`. Вычисление дельты занимает время $O(N)$, поскольку требуется пересечь два списка, содержащих $O(N)$ значений C, объединенных в пару с b в `S` и с a в `T` (т. е. кратность таких пар в `S` и `T` ненулевая).

Подход DBToaster ускоряет вычисление дельты за счет того, что заранее вычисляет три вспомогательных представления, соответствующих независимым от обновления частям дельта-запросов для обновления всех трех отношений:

```
CREATE VIEW V_ST(B, A, CNT) AS
  SELECT   B, A, SUM(P_S * P_T) as CNT
  FROM     S NATURAL JOIN T
  GROUP BY B, A

CREATE VIEW V_TR(C, B, CNT) AS
  SELECT   C, B, SUM(P_T * P_R) as CNT
  FROM     T NATURAL JOIN R
  GROUP BY C, B

CREATE VIEW V_RS(A, C, CNT) AS
  SELECT   A, C, SUM(P_R * P_S) as CNT
  FROM     R NATURAL JOIN S
  GROUP BY A, C
```

Представление V_ST позволяет вычислить дельта-запрос deltaQ за время $O(1)$, т. к. для соединения deltaR и V_ST нужен поиск пары (a, b) в V_ST, занимающий постоянное время. Но для обслуживания представлений V_RS и V_TR, определения которых включают R, все равно требуется время $O(N)$.

Метод F-IVM материализует только одно из трех представлений, например V_ST. В этом случае обслуживание при обновлениях R занимает время $O(1)$, но обслуживание S и T при обновлениях по-прежнему занимает время $O(N)$.

Алгоритм IVM$^\varepsilon$ классифицирует вершины графа по их степени, т. е. в зависимости от количества непосредственно соединенных с ними вершин: *тяжелыми* считаются вершины степени, большей или равной $N^{1/2}$, а *легкими* – вершины степени меньше $N^{1/2}$. Это ведет к разбиению каждой из трех копий R, S и T реберного отношения на тяжелую часть R_h(S_h, T_h) и легкую часть R_l(S_l, T_l): кортеж (a, b, p) принадлежит R_h, если a тяжелое, и R_l в противном случае. Аналогично кортеж (b, c, p) принадлежит S_h, если b тяжелое, и S_l в противном случае. Наконец, (c, a, p) принадлежит T_h, если c тяжелое, и T_l. Мы можем переписать Q, заменив каждое из трех отношений объединением двух его частей. Тогда запрос Q эквивалентен объединению восьми учитывающих асимметрию представлений Q_r,s,t, где $r, s, t \in \{h, l\}$:

```
CREATE VIEW Q_r,s,t(CNT) AS
  SELECT SUM(P_R * P_S * P_T) as CNT
  FROM   R_r NATURAL JOIN S_s NATURAL JOIN T_t
```

Рассмотрим затрагивающее один кортеж обновление deltaR_r = $\{(a, b, p)\}$ части R_r отношения R, где $r \in \{h, l\}$. Тогда вычисление дельты для представления Q_r,s,t описывается следующим более простым запросом:

```
CREATE VIEW deltaQ_r,s,t(CNT) AS
  SELECT SUM(P_R * P_S * P_T) as CNT
  FROM   deltaR_r NATURAL JOIN S_s NATURAL JOIN T_t
  WHERE  S_s.A = a AND T_t.B = b
```

IVM$^\varepsilon$ адаптирует свою стратегию обслуживания к каждому учитывающему асимметрию представлению, стремясь достичь сублинейного времени обновления. Хотя для большинства этих представлений тривиально достигается верхняя граница $O(N^{1/2})$, есть одно исключение. Далее мы объясним, как достичь этой границы при обслуживании каждого из этих представлений.

Вычисление дельты для четырех представлений Q_r,l,t (где $r, t \in \{h, l\}$) выражается следующим образом:

```
CREATE VIEW deltaQ_r,l,t(CNT) AS
  SELECT SUM(P_R * P_S * P_T) as CNT
  FROM   deltaR_r NATURAL JOIN S_l NATURAL JOIN T_t
  WHERE  S_l.A = a AND T_t.B = b
```

Части S_l и T_t соединяются по атрибуту C. Поскольку при обновлении deltaR_r производится присваивание B значения b в S_l, а b может быть только легким значением в S_l, то в пару с b в S_l соединено не больше $N^{1/2}$ значений C. Следовательно, для пересечения множества значений C в S_l и T_t потребуется время не больше $O(N^{1/2})$.

Вычисление дельты для представлений Q_r,h,h производится аналогично. Поскольку все значения C в T_h тяжелые, с каждым из них связано по крайней мере $N^{1/2}$ значений A. Это также означает, что существует не более $N^{1/2}$ тяжелых значений C. Следовательно, для пересечения множества тяжелых значений C в T_h и значений C в S_h потребуется время не больше $O(N^{1/2})$.

Однако для вычисления дельты для представлений Q_r,h,l, где $r \in \{h, l\}$, необходимо линейное время, т. к. требуется обойти все значения c, соединенные со значениями b в S_h и значениями a в T_l; количество таких значений C может линейно зависеть от размера базы данных. В таком случае IVM$^\varepsilon$ заранее вычисляет не зависящие от обновления части дельта-запросов в виде вспомогательных материализованных представлений, а затем использует их для ускорения вычисления дельты:

```
CREATE VIEW V_ST(B, A, CNT) AS
  SELECT   B, A, SUM(P_S * P_T) as CNT
  FROM     S_h NATURAL JOIN T_l
  GROUP BY B, A
```

В случае обновления T и S мы аналогично материализуем представления V_RS и V_TR соответственно. Для каждого из этих представлений нужно место размером $O(N^{3/2})$. Теперь можно вычислить deltaQ_r,h,l с помощью V_ST следующим образом:

```
CREATE VIEW deltaQ_r,h,l(CNT) AS
  SELECT SUM(P_R * CNT) as CNT
  FROM   deltaR_r NATURAL JOIN V_ST
  WHERE  V_ST.B = b AND V_ST.A = a
```

Это занимает время $O(1)$, поскольку нужно только произвести поиск в V_ST для получения кратности ребра (a, b), а затем умножить на p из deltaR_r.

3.2. Контроль доступа

Контроль доступа – важный аспект безопасности данных, функция СУБД, которая защищает данные от несанкционированного доступа. Еще один важный аспект – *защита данных*, т. е. воспрепятствование неавторизованным пользователям понять физическое содержание данных. В контексте централизованных и распределенных операционных систем эта функция обычно предоставляется файловой системой. Основной метод защиты данных – шифрование.

Механизм контроля доступа должен гарантировать, что только авторизованные пользователи могут выполнять операции в базе данных, и только те, которые им разрешены. Многие пользователи могут иметь доступ к большому набору данных, управляемому централизованной или распределенной системой. Поэтому СУБД, все равно, централизованная или распределенная, должна располагать средствами, которые позволяют ограничить доступ подмножеству пользователей к подмножеству данных. Долгое время

контроль доступа предоставлялся операционными системами как одна из функций файловой системы. В этом контексте контроль централизованный. Действительно центральный механизм создает объекты и может разрешить определенным пользователям выполнять над ними определенные операции (чтение, запись, исполнение). Кроме того, объекты идентифицируются внешними именами.

Контроль доступа в системах баз данных в нескольких отношениях отличается от традиционных файловых систем. Авторизацию следует уточнить, так чтобы разные пользователи могли иметь разные права доступа к одним и тем же объектам базы. Из этого требования вытекает необходимость описывать подмножества объектов более точно, чем по имени, и различать группы пользователей. Кроме того, в распределенном контексте особую важность приобретает децентрализованный контроль над авторизацией. В реляционных системах авторизация управляется администраторами базы данных с помощью высокоуровневых конструкций. Например, контролируемые объекты можно задавать такими же предикатами, как при формулировке запроса.

Существует два основных подхода к контролю доступа к базе данных. Первый называется *избирательное управление доступом* (discretionary access control – DAC) и уже давно предлагается СУБД. В этом случае определяются права доступа в зависимости от пользователя, типа доступа (например, **SELECT**, **UPDATE**) и объектов, к которым осуществляется доступ. Второй подход – *мандатное управление доступом* (mandatory access control – MAC) – повышает безопасность, предоставляя доступ к секретным данным только полномочным пользователям. Поддержка режима MAC в основных СУБД появилась сравнительно недавно в связи с новыми угрозами, исходящими от интернета. Имеются и другие подходы, основанные на привнесении дополнительной семантики, в частности ролевой контроль доступа, когда пользователям назначаются различные роли, и целевой контроль доступа, например в базах данных Гиппократа, в которых с данными ассоциируется информация о целях, т. е. о причинах сбора и доступа к данным.

Начав с решений в централизованных системах, мы перейдем к реализации контроля доступа в распределенных СУБД. Однако распределенность объектов и пользователей несет с собой дополнительную сложность. В следующих разделах мы сначала представим избирательный и мандатный контроль доступа в централизованных системах, а затем опишем дополнительные проблемы и их решения в распределенных системах.

3.2.1. Избирательный контроль доступа

В DAC-системах имеются три основные сущности: *субъект* (например, пользователь или группа пользователей), который инициирует выполнение прикладной программы, *операции*, производимые прикладной программой, и *объекты базы данных*, над которыми выполняются операции. Контроль авторизации – это проверка того, что тройке (субъект, операция, объект) разрешен доступ (т. е. пользователь может выполнить данную операцию с дан-

ным объектом). Авторизацию можно рассматривать как тройку (субъект, тип операции, определение объекта), которая постулирует, что у субъекта есть право выполнить операцию над объектом. Для управления авторизацией СУБД нуждается в определении субъектов, объектов и прав доступа.

Обычно субъект вводится в систему в виде пары (имя пользователя, пароль). Имя однозначно *идентифицирует* пользователя в системе, а пароль, известный только владельцу имени, *аутентифицирует* (доказывает подлинность) пользователя. Для входа в систему нужно указать и имя, и пароль. Это не даст войти пользователям, которые знают только имя, но не знают его пароля.

Подлежащие защите объекты – подмножество базы данных. Реляционные системы предоставляют более детальную и более общую схему защиты, чем предшествующие им. В файловой системе единицей защиты является файл, а в реляционной объекты можно определить как по типу (представление, отношение, кортеж, атрибут), так и по содержанию, пользуясь предикатами выборки. Кроме того, механизм представлений, описанный в разделе 3.1, позволяет защищать объекты, просто скрывая части отношений (атрибут или кортежи) от неавторизованных пользователей.

Право доступа выражает связь между субъектом и объектом для конкретного набора операций. В реляционных СУБД на основе SQL операцией считается высокоуровневая команда, например **SELECT**, **INSERT**, **UPDATE** или **DELETE**, а предоставление или отзыв прав задается соответственно командами

GRANT ⟨типы операций⟩ **ON** ⟨объект⟩ TO ⟨субъект(ы)⟩
REVOKE ⟨типы операций⟩ **FROM** ⟨объект⟩ TO ⟨субъект(ы)⟩

Ключевое слово *public* обозначает всех пользователей. Управление авторизацией можно охарактеризовать с точки зрения того, кто может предоставлять права. Чтобы облегчить администрирование базы данных, удобно определить группы пользователей, как в операционных системах, специально для целей авторизации. Определенная группа пользователей может выступать в роли субъекта в командах **GRANT** и **REVOKE**.

В простейшей форме контроль централизован: у одного пользователя или класса пользователей – администраторов базы данных – имеются все права доступа к объектам базы данных, и только им разрешено выполнять команды **GRANT** и **REVOKE**. В более общем случае контроль децентрализован: создатель объекта становится его владельцем и получает все права доступа к нему. В частности, существует дополнительный тип операции **GRANT** для передачи всех прав лица, выполнившего эту команду (праводателя), указанным субъектам. Поэтому лицо, получившее права (правополучатель), может впоследствии предоставлять права доступа к данному объекту. Таким образом, контроль доступа избирательный в том смысле, что пользователи, имеющие привилегию предоставлять права, могут принимать решения, касающиеся политики доступа. Процесс отзыва сложный, т. к. должен быть рекурсивным. Например, если *A*, предоставивший *B*, который предоставил *C* привилегию **GRANT** на объект *O*, захочет отозвать у *B* все права на *O*, то должны быть также отозваны все права *C* на *O*. Для выполнения отзыва система должна хранить всю иерархию предоставления прав на объект с создателем этого объекта в корне.

Права доступа субъектов к объектам хранятся в каталоге в виде правил авторизации. Существует несколько способов хранения таких правил. Самый удобный – рассматривать все привилегии как *матрицу авторизации*, в которой строка определяет субъект, столбец – объект, а элемент на пересечении строки и столбца содержит разрешенные операции. Разрешенные операции описываются типом (например, **SELECT**, **UPDATE**). С типом операции часто ассоциируют предикат, дополнительно ограничивающий доступ к объекту. Эта возможность используется, когда объекты должны быть базовыми отношениями, а не представлениями. Примером разрешенной операции для пары ⟨Jones, отношение EMP⟩ является команда

```
SELECT WHERE TITLE = "Syst.Anal."
```

которая разрешает пользователю Jones только доступ к кортежам работников, являющихся системными аналитиками. На рис. 3.5 приведен пример матрицы авторизации, в которой объектами являются отношения (EMP и ASG) или атрибуты (ENAME).

	EMP	ENAME	ASG
Casey	UPDATE	UPDATE	UPDATE
Jones	SELECT	SELECT	SELECT WHERE RESP ≠ "Manager"
Casey	NONE	SELECT	NONE

Рис. 3.5 ❖ Пример матрицы авторизации

Матрицу авторизации можно хранить тремя способами: по строкам, по столбцам или по элементам. Если матрица хранится по *строкам*, то с каждым субъектом ассоциирован список доступных ему объектов вместе с правами доступа к каждому. При таком подходе проверка авторизации осуществляется эффективно, потому что все права вошедшего в систему пользователя хранятся вместе (в профиле пользователя). Но манипуляция правами доступа к объекту (например, сделать объект публичным, т. е. доступным всем) затруднена, потому что приходится обращаться к профилям всех пользователей. Если матрица хранится по *столбцам*, то с каждым объектом ассоциирован список субъектов, имеющих к нему доступ, вместе с соответствующими правами доступа. Достоинства и недостатки этого подхода противоположны рассмотренному выше.

Достоинства обоих подходов можно объединить, если хранить матрицы по *элементам*, т. е. по отношению (субъект, объект, право). Это отношение можно проиндексировать по субъекту и объекту, обеспечив тем самым быстрый доступ к правам с обеих сторон.

Прямое управление связями между многими субъектами и объектами представляет трудную задачу для администраторов базы данных. *Ролевой контроль доступа* (role-based access control – RBAC) решает эту проблему путем добавления ролей – уровня независимости субъектов от объектов. Роли

соответствуют различным функциональным обязанностям (клерк, аналитик, менеджер и т. д.), а пользователи и права доступа к объектам ассоциируются с ролями. Таким образом, пользователь авторизуется не напрямую, а только через свои роли. Поскольку ролей не так много, RBAC заметно упрощает контроль доступа, в особенности при добавлении или изменении учетных записей пользователей.

3.2.2. Мандатный контроль доступа

У избирательного контроля доступа (DAC) имеются ограничения. Одна из проблем заключается в том, что злонамеренный пользователь может получить несанкционированный доступ к данным через авторизованного пользователя. Рассмотрим, к примеру, пользователя *A*, которому разрешен доступ к отношениям `R` и `S`, и пользователя *B*, которому разрешен доступ только к отношению `S`. Если *B* каким-то образом удастся модифицировать прикладную программу, используемую *A*, так что она будет записывать данные `R` в `S`, то *B* сможет читать данные, которые должны быть ему недоступны, не нарушая правил авторизации.

MAC решает эту проблему и повышает защищенность системы путем определения различных уровней безопасности для субъектов и объектов данных. Кроме того, в отличие от DAC, решения, касающиеся политики доступа, принимаются только администратором, т. е. пользователи не могут определять собственные политики и предоставлять доступ к объектам. В базах данных MAC основан на хорошо известной модели Белла–Лападулы, первоначально спроектированной для обеспечения безопасности операционной системы. В этой модели субъектами являются процессы, действующие от имени пользователя; с процессом ассоциирован уровень безопасности, или *допуск* (гриф секретности), выведенный на основе пользователя. В простейшей форме имеются четыре уровня безопасности: совершенно секретно (*TS*), секретно (*S*), конфиденциально (*C*) и несекретно (*U*) – упорядоченные следующим образом: $TS > S > C > U$, где «>» означает «более безопасно». Доступ со стороны субъектов в режимах чтения и записи ограничен двумя простыми правилами:

1) субъекту *T* разрешено читать объект с грифом секретности *l*, только если $level(T) \geqslant l$;
2) субъекту *T* разрешено записывать объект с грифом секретности *l*, только если $level(T) \leqslant l$.

Правило 1 («запрет чтения вверх») защищает данные от несанкционированного раскрытия, т. е. субъект с данным уровнем безопасности может читать только объекты, находящиеся на том же или более низком уровне безопасности. Например, субъект с допуском «секретно» не может читать совершенно секретные данные. Правило 2 («запрет записи вниз») защищает данные от несанкционированного изменения, т. е. субъект с некоторым уровнем безопасности может записывать только объекты, находящиеся на том же или более высоком уровне безопасности. Например, субъект с допуском «совершенно секретно» может записывать совершенно секретные данные,

но не может записывать секретные данные (поскольку тогда они могли бы содержать совершенно секретные части).

В реляционной модели объектами данных могут быть отношения, кортежи или атрибуты. Следовательно, отношение можно классифицировать на разных уровнях: отношение (т. е. все кортежи имеют один и тот же уровень безопасности), кортеж (т. е. у каждого кортежа свой уровень безопасности) или атрибут (т. е. у каждого атрибута свой уровень безопасности). Классифицированное отношение поэтому называется *многоуровневым отношением*, чтобы отразить тот факт, что оно выглядит по-разному (содержит разные данные) для субъектов с разными допусками. Например, многоуровневое отношение, классифицированное на уровне кортежей, можно представить, добавив соответствующий уровень безопасности к каждому атрибуту. На рис. 3.6 показано многоуровневое отношение PROJ*, основанное на отношении PROJ и классифицированное на уровне атрибутов. Заметим, что дополнительный уровень безопасности атрибутов может значительно увеличить размер отношения.

PROJ*

PNO	SL1	PNAME	SL2	BUDGET	SL3	LOC	SL4
P1	C	Instrumentation	C	150000	C	Montreal	C
P2	C	Database Develop.	C	135000	S	New York	S
P3	S	CAD/CAM	S	250000	S	New York	S

Рис. 3.6 ❖ Многоуровневое отношение PROJ*, классифицированное на уровне атрибутов

Отношение в целом также имеет уровень безопасности, равный наименьшему уровню безопасности хранящихся в нем данных. Например, уровень безопасности отношения PROJ* равен *C*. К отношению может обратиться любой субъект с таким же или более высоким уровнем безопасности. Однако субъекту будут доступны только данные, к которым у него есть допуск. Атрибуты, к которым у субъекта нет допуска, будут выглядеть для него как null-значения с уровнем безопасности таким же, как у самого субъекта. На рис. 3.7 показан пример отношения PROJ*, каким оно представляется субъекту с уровнем безопасности «конфиденциально».

PROJ*C

PNO	SL1	PNAME	SL2	BUDGET	SL3	LOC	SL4
P1	C	Instrumentation	C	150000	C	Montreal	C
P2	C	Database Develop.	C	Null	S	Null	S

Рис. 3.7 ❖ Конфиденциальное отношение PROJ*C

MAC оказывает сильное влияние на модель данных, потому что пользователи видят разные данные и сталкиваются с неожиданными побочными эффектами. Один такой эффект называется *многоэкземплярностью* (polyin-

stantiation), это означает, что у одного и того же объекта могут быть разные значения атрибутов в зависимости от уровня безопасности пользователя. На рис. 3.8 показано многоуровневое отношение с многоэкземплярными кортежами. У кортежа с первичным ключом P3 два экземпляра, по одному для каждого уровня безопасности. Такое может случиться, если субъект *T* с уровнем безопасности *C* вставит кортеж с ключом P3 в отношение PROJ* на рис. 3.6. Поскольку *T* (с конфиденциальным допуском) должен игнорировать существование кортежа с ключом P3 (классифицированным как секретный), единственное практически осуществимое решение – добавить второй кортеж с таким же ключом и другой классификацией. Однако пользователь с секретным допуском будет видеть оба кортежа с ключом E3 и должен как-то интерпретировать этот неожиданный эффект.

PROJ**

PNO	SL1	PNAME	SL2	BUDGET	SL3	LOC	SL4
P1	C	Instrumentation	C	150000	C	Montreal	C
P2	C	Database Develop.	C	135000	S	New York	S
P3	S	CAD/CAM	S	250000	S	New York	S
P3	C	Web Develop.	C	200000	C	Paris	C

Рис. 3.8 ❖ Многоуровневое отношение с многоэкземплярностью

3.2.3. Распределенный контроль доступа

Дополнительные проблемы контроля доступа в распределенной среде связаны с тем, что объекты и субъекты распределены и сообщения, содержащие секретные данные, могут быть прочитаны неавторизованными пользователями. Вот эти проблемы: удаленная аутентификация пользователей, управление правилами избирательного доступа, обработка представлений и групп пользователей, реализация MAC.

Удаленная аутентификация пользователей необходима, потому что любой узел распределенной СУБД может принимать запросы от программ, запущенных и авторизованных в удаленном узле. Чтобы предотвратить удаленный доступ со стороны неавторизованных пользователей или приложений (например, из узла, не являющегося частью распределенной СУБД), пользователей необходимо аутентифицировать также в узле, к которому они обращаются. Кроме того, вместо паролей, которые можно перехватить, прослушивая сообщения, необходимо использовать зашифрованные сертификаты.

Есть три решения для управления аутентификацией:

1) информация, необходимая для аутентификации, хранится в центральном узле для *глобальных пользователей*, которые могут быть аутентифицированы только один раз и затем обращаться к разным узлам;
2) информация для аутентификации пользователей (имя пользователя и пароль) реплицируется на все узлы каталога. Локальные программы, запущенные в удаленном узле, также должны сообщать имя и пароль пользователя;

3) все узлы распределенной СУБД аутентифицируют себя так же, как это делают пользователи. Таким образом, межузловой обмен данными защищен паролем узла. После того как узел-инициатор аутентифицирован, дополнительно аутентифицировать удаленных пользователей, работающих на нем, не нужно.

Первое решение сильно упрощает администрирование паролей и допускает единую аутентификацию (или, как говорят, единую точку входа). Однако центральный узел аутентификации может стать точкой общего отказа или узким местом. Второе решение дороже с точки зрения управления каталогом, учитывая, что добавление нового пользователя – распределенная операция. Однако пользователи могут обращаться к распределенной базе данных из любого узла. Третье решение необходимо, если информация о пользователях не реплицируется. Но его можно использовать и при наличии репликации. В таком случае удаленная аутентификация становится более эффективной. Если имена и пароли пользователей не реплицируются, то их следует хранить в узлах, с которых пользователи обращаются к системе (домашних узлах). Последнее решение основано на реалистичном предположении, что пользователи достаточно статичны или, по крайней мере, обращаются к распределенной базе данных из одного и того же узла.

Распределенные правила авторизации выражаются так же, как централизованные. Как и определения представлений, они должны храниться в каталоге. Их можно либо полностью реплицировать в каждом узле, либо хранить в узлах, где находятся упоминаемые объекты. В последнем случае правила дублируются только в узлах, на которые распределены упоминаемые в них объекты. Главное преимущество полной репликации в том, что авторизацию можно выполнить путем модификации запроса на этапе компиляции. Но из-за дублирования данных управление каталогом усложняется. Второе решение лучше, если локальность ссылок очень высока. Однако контролировать распределенную авторизацию на этапе компиляции невозможно.

Механизм авторизации может трактовать представления как объекты. Представления – это составные объекты, составленные из объектов более низкого уровня. Поэтому предоставление доступа к представлению транслируется в предоставление доступа к этим объектам. Если определения представлений и правила авторизации для всех объектов полностью реплицированы (как во многих системах), то эта трансляция сравнительно проста и может быть выполнена локально. Сложнее, если определения представления и составляющих его объектов хранятся порознь, как в случае предположения об автономности узла. В таком случае трансляция – полностью распределенная операция. Права доступа к представлениям зависят от того, какие права имеет создатель представления на доступ к составляющим его объектам. Решение – хранить информацию об ассоциации в узле каждого составляющего объекта.

Применение групп пользователей для целей авторизации упрощает администрирование распределенной базы данных. В централизованной СУБД «все пользователи» составляют специальную группу *public*. В распределенной СУБД это понятие тоже полезно, public обозначает всех пользователей системы. Но часто используется также промежуточный уровень – все пользователи

в одном узле, эта группа обозначается, например, public@site_s. Более точные группы определяются командой

```
DEFINE GROUP ⟨group_id⟩ AS ⟨список_идентификаторов_субъектов⟩
```

Управление группами в распределенной среде поднимает новые проблемы, потому что субъекты группы могут находиться в разных узлах, а доступ к объекту может предоставляться нескольким группам, которые сами распределены. Если информация о группах и правила доступа полностью реплицированы во всех узлах, то проверка прав доступа производится так же, как в централизованной системе. Но поддерживать такую репликацию дорого. Проблема еще осложняется, если требуется поддерживать автономность узлов (с децентрализованным контролем). Один из способов проверки прав доступа состоит в том, чтобы выполнять удаленный запрос к узлам, где хранится определение группы. Другое решение – реплицировать определение группы в каждом узле, содержащем объекты, к которым могут обращаться субъекты из этой группы. Но эти решения снижают степень автономности узла.

Реализация MAC в распределенной среде осложняется возможностью несанкционированного доступа к данным с помощью косвенных средств, называемых *скрытыми каналами*. Например, рассмотрим простую архитектуру распределенной СУБД с двумя узлами, каждый из которых управляет базой данных с одним уровнем безопасности, например в одном узле хранятся конфиденциальные данные, а в другом секретные. Согласно правилу запрета «записи вниз», операция обновления, исходящая от субъекта с секретным допуском, может быть адресована только секретному узлу. Но, по правилу запрета «чтения вверх», запрос чтения от того же секретного субъекта может быть адресован как секретному, так и конфиденциальному узлу. Поскольку запрос, адресованный конфиденциальному узлу, может содержать секретную информацию (например, в предикате выборки), потенциально он является скрытым каналом. Чтобы избежать таких скрытых каналов, нужно реплицировать часть базы данных, так чтобы узел с уровнем безопасности l содержал все данные, к которым может обращаться субъект на уровне l. Например, секретный узел должен был бы реплицировать конфиденциальные данные, так чтобы он мог обработать секретные запросы целиком. Проблема этой архитектуры – в накладных расходах на поддержание согласованности реплик (о репликации см. главу 6). Кроме того, даже если нет скрытых каналов для запросов, могут существовать скрытые каналы для операций обновления, т. к. можно воспользоваться информацией о задержках при синхронизации транзакций. Поэтому для полной поддержки MAC в распределенных СУБД требуется значительно расширить методы управления транзакциями и обработки распределенных запросов.

3.3. Контроль семантической целостности

Еще одна важная и трудная проблема для системы баз данных – как гарантировать *согласованность базы данных*. Говорят, что база данных на-

ходится в согласованном состоянии, если она удовлетворяет множеству *ограничений семантической целостности*. Для поддержания базы данных в согласованном состоянии необходимы различные механизмы, в т. ч. управление конкурентностью, надежность, защита и контроль семантической целостности; все они являются частью управления транзакциями. Контроль семантической целостности гарантирует согласованность базы данных, отвергая те транзакции обновления, которые переводят базу данных в несогласованное состояние, или инициируя некоторые действия, которые компенсируют последствия транзакций обновления. Заметим, что обновленная база данных должна удовлетворять набору ограничений целостности.

В общем случае ограничения семантической целостности – это правила, представляющие *знания* о свойствах приложения. Они определяют статические или динамические свойства, которые невозможно непосредственно отразить в самом объекте и в концепциях операции, принятых в модели данных. Таким образом, понятие правила целостности тесно связано с понятием модели данных в том смысле, что с помощью этих правил можно уловить больше семантической информации о приложении.

Можно выделить два главных типа ограничений целостности: структурные и поведенческие. *Структурные ограничения* выражают основные семантические свойства, внутренне присущие модели. Примерами могут служить ограничения уникального ключа в реляционной модели или ассоциации один-ко-многим между объектами в объектно-ориентированной модели. С другой стороны, *поведенческие ограничения* регулируют поведение приложения, а значит, неотъемлемы от процесса проектирования базы данных. Они могут выражать ассоциации между объектами, например зависимость по включению в реляционной модели, или описывать свойства и структуру объектов. Расширяющееся многообразие приложений баз данных и разработка инструментов для проектирования баз данных требуют мощных ограничений целостности, способных обогатить модель данных.

Контроль целостности возник одновременно с обработкой данных и развивался в направлении от процедурных методов (когда средства контроля включены в прикладные программы) к декларативным. Декларативные методы появились вместе с реляционной моделью и были призваны сгладить проблемы зависимости от программ и данных, избыточности кода и низкой производительности процедурных методов. Идея в том, чтобы выразить ограничения целостности с помощью конструкций исчисления предикатов. Таким образом, набор утверждений о семантической целостности определяет согласованность базы данных. Этот подход позволяет легко объявлять и модифицировать сложные ограничения целостности.

Главная проблема поддержки автоматического контроля семантической целостности заключается в том, что стоимость проверки нарушения ограничений может оказаться непомерно высокой. Проверка ограничений целостности стоит дорого, потому что в общем случае требует доступа к большому объему данных, не относящемуся непосредственно к операции обновления. Проблема еще осложняется, когда ограничения определены в распределенной базе данных.

Были исследованы различные способы проектирования менеджера целостности путем сочетания стратегий оптимизации. Их цель – (1) лимитировать количество проверяемых ограничений; (2) – уменьшить количество операций доступа, необходимых для проверки заданного ограничения при наличии транзакции обновления; (3) определить превентивную стратегию, которая обнаруживает несогласованность, не выполняя обновлений; (4) выполнять как можно большую часть контроля целостности на этапе компиляции запроса. Некоторые из этих решений были реализованы, но им недостает общности. Либо они распространяются на слишком узкое множество ограничений (стоимость проверки более общих ограничений была бы недопустимо высока), либо поддерживают только ограниченный набор программ (например, обновления одного кортежа).

В этом разделе мы сначала опишем подходы к контролю семантической целостности в централизованных базах данных, а затем перейдем к распределенным. Поскольку обсуждение ведется в контексте реляционной модели, рассматриваются только декларативные методы.

3.3.1. Централизованный контроль семантической целостности

Менеджер семантической целостности состоит из двух основных компонент: язык для выражения и манипулирования ограничениями целостности и механизм проверки, который предпринимает определенные действия для поддержания целостности базы данных при выполнении транзакций обновления.

3.3.1.1. Спецификация ограничений целостности

Ограничения целостности задаются администратором базы данных на языке высокого уровня. В этом разделе мы представим декларативный язык для описания ограничений целостности. Он во многом следует духу стандартного SQL, но обладает большей общностью. Он позволяет создавать, читать и удалять ограничения целостности. Ограничения можно определять как в момент создания отношения, так и в любое другое время, даже если отношение уже содержит кортежи. Синтаксис в обоих случаях почти одинаковый. Для простоты и без потери общности будем предполагать, что при нарушении ограничения целостности транзакция отменяется. Однако стандарт SQL предлагает средства, позволяющие выразить распространение обновлений для корректировки несогласованности – фразу `CASCADING` в объявлении ограничения. Можно использовать и более общий механизм *триггеров* (правила вида событие–условие–действие) для автоматического распространения обновлений и тем самым поддержания семантической целостности. Но триггеры – это чрезвычайно мощное средство, которое труднее поддержать эффективно, чем конкретные ограничения целостности.

В реляционных базах данных ограничения целостности определяются в виде утверждений. Утверждение – это конкретное выражение реляционно-

го исчисления кортежей, в котором каждой переменной предшествует квантор всеобщности (∀) или существования (∃). Таким образом, утверждение можно рассматривать как описание запроса, который принимает значение true или false для каждого кортежа в декартовом произведении отношений, определенных кортежными переменными. Можно выделить три типа ограничений целостности: предопределенные, предусловие и общего вида.

Предопределенные ограничения основаны на простых ключевых словах. С их помощью мы можем кратко выразить такие распространенные ограничения реляционной модели, как не null, уникальный ключ, внешний ключ или функциональную зависимость. В примерах 3.11–3.14 демонстрируются предопределенные ограничения.

Пример 3.11. Номер работника в отношении EMP не может быть null.

```
ENO NOT NULL IN EMP
```
♦

Пример 3.12. Пара (ENO, PNO) является уникальным ключом отношения ASG.

```
(ENO, PNO) UNIQUE IN ASG
```
♦

Пример 3.13. Номер проекта PNO в отношении ASG – внешний ключ, соответствующий первичному ключу PNO отношения PROJ. Иными словами, проект, встречающийся в отношении ASG, должен существовать в отношении PROJ.

```
PNO IN ASG REFERENCES PNO IN PROJ
```
♦

Пример 3.14. Номер работника функционально определяет имя работника.

```
ENO IN EMP DETERMINES ENAME
```
♦

Ограничения предусловия выражают условия, которые должны удовлетворяться для всех кортежей отношения при обновлении заданного типа. Тип обновления – **INSERT**, **DELETE** или **MODIFY** – позволяет ограничить контроль целостности. Чтобы описать в определении ограничения кортежи, подлежащие обновлению, неявно определяются две переменные, NEW и OLD, областью значений которых являются соответственно новые кортежи (вставляемые) и старые кортежи (удаляемые). Ограничения предусловия можно выразить с помощью SQL-команды **CHECK**, дополненной возможностью задать тип обновления. Синтаксис команды **CHECK** такой:

```
CHECK ON ⟨имя отношения⟩ WHEN ⟨тип изменения⟩
  (⟨квалификация кортежей в отношении⟩)
```

Ниже приведены примеры ограничений предусловия.

Пример 3.15. Бюджет проекта заключен между 500K и 1000K.

```
CHECK ON PROJ (BUDGET+ >= 500000 AND BUDGET <= 1000000)
```
♦

Пример 3.16. Разрешается удалять только кортежи, в которых бюджет равен 0.

```
CHECK ON PROJ WHEN DELETE (BUDGET = 0)
```
♦

Пример 3.17. Бюджет проекта может только увеличиваться.

```
CHECK ON PROJ (NEW.BUDGET > OLD.BUDGET AND
  NEW.PNO = OLD.PNO)                                    ♦
```

Ограничения общего вида – это формулы реляционного исчисления кортежей, в которых все переменные квантифицированы. Система баз данных должна гарантировать, что эти формулы всегда истинны. Ограничения общего вида более лаконичны, чем предкомпилированные ограничения, поскольку могут включать более одного отношения. Например, для выражения ограничения общего вида с тремя отношениями необходимо как минимум три предкомпилированных ограничения. Для записи ограничений общего вида применяется следующий синтаксис:

```
CHECK ON список ⟨имя переменной⟩:⟨имя отношения⟩,
  (⟨квалификация⟩)
```

Ниже приведены примеры ограничений общего вида.

Пример 3.18. Ограничение из примера 3.8 можно записать также в виде:

```
CHECK ON e1:EMP, e2:EMP
  (e1.ENAME = e2.ENAME IF e1.ENO = e2.ENO)              ♦
```

Пример 3.19. Суммарное время работы всех работников над проектом CAD меньше 100.

```
CHECK ON g:ASG, j:PROJ (SUM(g.DUR WHERE
  g.PNO=j.PNO)<100 IF j.PNAME="CAD/CAM")                ♦
```

3.3.1.2. Проверка целостности

Теперь мы сосредоточимся на поддержании семантической целостности, которая заключается в том, чтобы отвергать транзакции обновления, нарушающие некоторые ограничения целостности. Ограничение считается нарушенным, если оно принимает значение false в новой базе данных, порожденной транзакцией обновления. Главная трудность при проектировании менеджера целостности – придумать эффективные алгоритмы проверки. Есть два основных метода отвергнуть несогласованные транзакции обновления. Первый основан на *обнаружении* несогласованностей. Транзакция обновления u выполняется, и это переводит базу данных из состояния D в состояние D_u. Алгоритм применяет проверки, выведенные из ограничений, и убеждается, что в состоянии D_u соблюдены все релевантные ограничения. Если состояние D_u несогласованное, то СУБД может либо перейти в другое согласованное состояние D'_u, модифицировав D_u с помощью компенсационных действий, либо восстановить состояние D, откатив u. Поскольку эти тесты применяются *после* того, как состояние базы данных изменено, они называются *апостериорными тестами*. Этот подход может оказаться неэффективным, если большой объем проделанной работы (обновление D) необходимо откатить в случае нарушения ограничения.

Второй метод основан на *предотвращении* несогласованностей. Обновление выполняется, только если изменения переводят базу данных в согласованное состояние. Кортежи, являющиеся предметом транзакции обновления, либо доступны непосредственно (в случае вставки), либо должны быть извлечены из базы (в случае удаления или модификации). Алгоритм проверяет, что все релевантные ограничения будут соблюдены после обновления этих кортежей. Обычно для этого к кортежам применяются тесты, выведенные из ограничений целостности. Поскольку эти тесты применяются *до* того, как состояние базы данных изменилось, они называются *априорными*. Превентивный подход эффективнее обнаружения, так как в случае нарушения целостности обновления не придется откатывать.

Алгоритм модификации запроса – пример превентивного метода, который особенно эффективен при проверке ограничений области определения. Он дополняет квалификацию запроса утверждением, присоединяемым с помощью оператора AND, так что модифицированный запрос включает проверку ограничения.

Пример 3.20. Запрос увеличения бюджета проекта CAD/CAM на 10 %, который можно было бы записать в виде

```
UPDATE PROJ
SET    BUDGET = BUDGET*1.1
WHERE  PNAME= "CAD/CAM"
```

будет преобразован в следующий запрос, чтобы гарантировать выполнение ограничения области определения, которое обсуждалось в примере 3.9:

```
UPDATE PROJ
SET    BUDGET = BUDGET * 1.1
WHERE  PNAME= "CAD/CAM"
AND    NEW.BUDGET ≥ 500000
AND    NEW.BUDGET ≤ 1000000
```

♦

Алгоритм модификации запроса, хорошо известный своей элегантностью, порождает априорные тесты на этапе выполнения, присоединяя с помощью AND предикаты утверждения к предикатам обновления в каждой команде транзакции. Однако этот алгоритм применим только к формулам исчисления кортежей, и описать его можно следующим образом. Рассмотрим утверждение $(\forall x \in \mathtt{R})F(x)$, где F – выражение исчисления кортежей, в котором x – единственная свободная переменная. Обновление R можно записать в виде $(\forall x \in \mathtt{R})(Q(x) \Rightarrow update(x))$, где Q – выражение исчисления кортежей, в котором x – единственная свободная переменная. Грубо говоря, модификация кортежей сводится к порождению обновления $(\forall x \in \mathtt{R})((Q(x)$ и $F(x)) \Rightarrow update(x))$. Таким образом, переменной x должен предшествовать квантор всеобщности.

Пример 3.21. Ограничение внешнего ключа из примера 3.13, которое можно переписать в виде

$\forall g \in \mathtt{ASG},\ \exists j \in \mathtt{PROJ} : g.\mathtt{PNO} = j.\mathtt{PNO}$

нельзя было бы обработать с помощью модификации запроса, поскольку переменной *j* не предшествует квантор всеобщности. ♦

Для обработки более общих ограничений априорные тесты можно генерировать в момент определения ограничения и проверять на этапе выполнения, когда производится обновление. Далее в этом разделе мы представим общий метод. Идея заключается в том, чтобы в момент определения ограничения не порождать априорные тесты, которые впоследствии можно будет использовать, дабы предотвратить появление несогласованностей в базе данных. Это общий превентивный метод, применимый ко всем ограничениям, описанным в предыдущем разделе. Он заметно уменьшает часть базы данных, подлежащую просмотру при проверке ограничений после обновления. В случае распределенной среды это важное преимущество.

В определении априорного теста используются дифференциальные отношения, определенные в разделе 3.1.3. *Априорный тест* – это тройка (R, *U*, *C*), где R – отношение, *U* – тип обновления, а *C* – утверждение над дифференциальными отношениями, связанными с обновлением типа *U*. При определении ограничения целостности *I* необходимо создать набор априорных тестов для отношений, встречающихся в *I*. Когда отношение, встречающееся в *I*, обновляется транзакцией *u*, проверке подлежат только априорные тесты, определенные над *I* для обновления типа *u*. У этого подхода двоякое преимущество с точки зрения производительности. Во-первых, количество проверяемых утверждений минимизируется, т. к. проверять нужно только априорные тесты типа *u*. Во-вторых, стоимость проверки априорного теста меньше, чем стоимость проверки *I*, т. к. дифференциальные отношения, вообще говоря, намного меньше базовых. Априорные тесты можно получить, применив правила преобразования к исходному утверждению. Эти правила основаны на синтаксическом анализе утверждения и перестановках кванторов. Они допускают подстановку дифференциальных отношений вместо базовых. Поскольку априорные тесты проще исходных, процесс их порождения называется *упрощением*.

Пример 3.22. Рассмотрим модифицированное выражение ограничения внешнего ключа в примере 3.15. С этим ограничением ассоциированы априорные тесты

(ASG, **INSERT**, C_1), (PROJ, **DELETE**, C_2) и (PROJ, **MODIFY**, C_3),

где C_1 имеет вид

$\forall$NEW $\in$ ASG$^+$, $\exists j \in$ PROJ: NEW.PNO = *j*.PNO;

C_2 имеет вид

$\forall g \in$ ASG, $\forall$OLD $\in$ PROJ$^-$: *g*.PNO $\neq$ OLD.PNO;

а C_3 имеет вид

$\forall g \in$ ASG, $\forall$OLD $\in$ PROJ$^-$ $\exists$NEW $\in$ PROJ$^+$: *g*.PNO $\neq$ OLD.PNO OR
OLD.PNO = NEW.PNO. ♦

Преимущества таких априорных тестов очевидны. Например, удаление из отношения ASG не влечет за собой проверки каких-либо утверждений.

Алгоритм проверки ограничений использует априорные тесты и специализирован в зависимости от класса утверждений. Выделяется три класса ограничений: с одним отношением, с несколькими отношениями и с участием агрегатных функций.

Резюмируем алгоритм проверки. Напомним, что транзакция обновления изменяет все кортежи отношения R, которые удовлетворяют некоторому условию. Алгоритм состоит из двух шагов. На первом шаге по R порождаются дифференциальные отношения R^+ и R^-. На втором шаге из R^+ и R^- просто выбираются кортежи, не удовлетворяющие априорным тестам. Если таких кортежей не оказалось, значит, ограничение удовлетворяется. В противном случае оно нарушено.

Пример 3.23. Предположим, что производится удаление из PROJ. Проверка ограничения (PROJ, **DELETE**, C_2) включает порождение следующей команды:

результат ← выбрать все кортежи из $PROJ^-$, где $\neg(C_2)$

Если результат пуст, то ограничение успешно проверено и согласованность сохраняется. ♦

3.3.2. Распределенный контроль семантической целостности

В этом разделе мы представим алгоритмы поддержания семантической целостности распределенных баз данных. Все они являются обобщениями рассмотренного выше метода упрощения. Далее предполагается, что имеется глобальный механизм управления транзакциями, как в однородных или мультибазовых системах. Поэтому две основные проблемы проектирования менеджера целостности для такой распределенной СУБД – как определить и хранить ограничения и как их проверять. Мы также обсудим проблемы проверки ограничений целостности, возникающие, когда глобальной поддержки транзакций нет.

3.3.2.1. Определение распределенных ограничений целостности

Предполагается, что ограничение целостности выражено средствами исчисления предикатов. Каждое утверждение рассматривается как условие запроса, которое истинно или ложно для любого кортежа, принадлежащего декартову произведению отношений, определенных кортежными переменными. Поскольку утверждения могут относиться к данным, хранящимся в разных узлах, необходимо решить, как хранить ограничения целостности, чтобы минимизировать стоимость их проверки. Существует стратегия, основанная на выделении трех классов ограничений.

1. *Индивидуальные ограничения*: ограничения с одним отношением и одной переменной. Они распространяются только на кортежи, обновляемые независимо от остальной базы данных. Например, ограничение области определения из примера 3.15 является индивидуальным.
2. *Ограничения, ориентированные на обработку множеств*: включают ограничения с одним отношением и несколькими переменными, например функциональные зависимости (пример 3.14), и ограничения с несколькими отношениями и несколькими переменными, например ограничения внешнего ключа (пример 3.13).
3. *Ограничения с агрегатами*: требуют специальной обработки из-за стоимости вычисления агрегатов. Утверждение из примера 3.19 – представитель этого класса.

Определение нового ограничения целостности можно начать в каком-то узле, где хранятся участвующие в утверждении отношения. Напомним, что отношения могут быть фрагментированы. Предикат фрагментации – частный случай утверждения класса 1. Разные фрагменты одного и того же отношения могут находиться в разных узлах. Таким образом, определение ограничения целостности становится распределенной операцией, состоящей из двух шагов. На первом шаге высокоуровневые утверждения преобразуются в априорные тесты, для чего применяются методы, описанные в предыдущем разделе. На следующем шаге априорные тесты сохраняются с учетом класса ограничений. Ограничения класса 3 обрабатываются как ограничения класса 1 или 2 в зависимости от того, являются они индивидуальными или ориентированными на обработку множеств.

Индивидуальные ограничения

Определение ограничения рассылается все узлам, содержащим фрагменты участвующего в ограничении отношения. Ограничение должно быть совместимо с данными отношения в каждом узле. Совместимость проверяется на двух уровнях: предиката и данных. Сначала проверяется совместимость предикатов путем сравнения предиката ограничения с предикатом фрагмента. Ограничение *C* считается несовместимым с предикатом фрагмента *p*, если из утверждения «*C* истинно» вытекает, что «*p* ложно», в противном случае они совместимы. Если в каком-то узле обнаружена несовместимость, то определение ограничения глобально отвергается, потому что кортежи, принадлежащие этому фрагменту, не удовлетворяют ограничению целостности. Затем, если установлена совместимость предикатов, ограничение проверяется на совместимость с данными фрагмента. Если фрагмент ему не удовлетворяет, то ограничение глобально отвергается. Если совместимость подтверждена, то ограничение сохраняется в каждом узле. Заметим, что проверки совместимости производятся только для априорных тестов с типом обновления «вставка» (кортежи, принадлежащие фрагментам, считаются «вставленными»).

Пример 3.24. Рассмотрим отношение `EMP`, горизонтально фрагментированное на три узла с использованием предикатов

p_1: $0 \leqslant$ `ENO` < "E3"

p_2: "E3" ⩽ ENO ⩽ "E6"
p_3: ENO > "E6"

и ограничение области определения C: ENO < "E4". Ограничение C совместимо с p_1 (если C истинно, то p_1 истинно) и p_2 (если C истинно, то p_2 необязательно ложно), но не с p_3 (если C истинно, то p_3 ложно). Поэтому ограничение C следует глобально отвергнуть, т. к. кортежи в узле 3 ему не удовлетворяют, так что отношение EMP не удовлетворяет C. ♦

Ограничения, ориентированные на обработку множеств

Ограничения, ориентированные на обработку множеств, содержат несколько переменных, т. е. включают предикаты соединения. Хотя предикат утверждения может включать несколько отношений, априорный тест ассоциируется с одним отношением. Поэтому определение ограничения можно разослать во все узлы, где хранится фрагмент, на который ссылаются переменные. Проверка совместимости также включает фрагменты отношения, используемого в предикате соединения. Здесь проверка совместимости предикатов бесполезна, потому что невозможно установить, что предикат фрагмента p ложен, если ограничение C (основанное на предикате соединения) истинно. Поэтому C следует проверять на совместимость с данными. Для этой проверки требуется соединить каждый фрагмент отношения, скажем R, со всеми фрагментами другого отношения, скажем S, упоминаемого в предикате ограничения. Эта операция может оказаться дорогостоящей и, как любое соединение, должна быть оптимизирована процессором распределенных запросов. Возможны три случая – в порядке увеличения стоимости проверки.

1. Фрагментация R является производной (см. главу 2) от фрагментации S и основана на полусоединении по атрибуту, участвующему в предикате соединения в утверждении.
2. S фрагментировано по атрибуту соединения.
3. S не фрагментировано по атрибуту соединения.

В первом случае проверка совместимости обходится дешево, потому что кортеж S, соответствующий кортежу R, находится в том же узле. Во втором случае каждый кортеж R необходимо сравнить самое большее с одним фрагментом S, потому что значением атрибута соединения в кортеже R можно воспользоваться для нахождения узла, где находится соответствующий фрагмент S. В третьем случае каждый кортеж необходимо сравнить со всеми фрагментами S. Если совместимость установлена для всех кортежей R, то ограничение можно сохранить в каждом узле.

Пример 3.25. Рассмотрим априорный тест, ориентированный на обработку множеств (ASG, **INSERT**, C_1), определенный в примере 3.16, где C_1 имеет вид

$\forall$NEW $\in$ ASG$^+$, $\exists j \in$ PROJ: NEW.PNO = j.PNO.

Рассмотрим следующие три случая.

1. ASG фрагментировано по предикату

   $\text{ASG} \ltimes_{\text{PNO}} \text{PROJ}_i$,

где $PROJ_i$ – фрагмент отношения PROJ. В этом случае каждый кортеж NEW отношения ASG был размещен в том же узле, что кортеж *j*, для которого NEW.PNO = *j*.PNO. Так как предикат фрагментации идентичен предикату C_1, для проверки совместимости никакой обмен данными по сети не нужен.

2. PROJ горизонтально фрагментировано по двум предикатам

   p_1: PNO < "P3"
   p_2: PNO ⩾ "P3"

   В этом случае каждый кортеж NEW отношения ASG сравнивается либо с фрагментом $PROJ_1$, если NEW.PNO < "P3", либо с фрагментом $PROJ_2$, если NEW.PNO ⩾ "P3".
3. PROJ горизонтально фрагментировано по двум предикатам

   p_1: PNAME = "CAD/CAM"
   p_2: PNAME ≠ "CAD/CAM"

   В этом случае каждый кортеж ASG необходимо сравнить с обоими фрагментами $PROJ_1$ и $PROJ_2$. ♦

3.3.2.2. Проверка распределенных ограничений целостности

Проверять распределенные ограничения целостности труднее, чем в централизованных СУБД даже при наличии поддержки глобального управления транзакциями. Основная проблема – решить, где (в каком узле) проверять ограничения целостности. Выбор зависит от класса ограничения, от типа обновления и от природы узла, инициировавшего обновление (он называется *главным узлом запроса*). Обновленное отношение или некоторые отношения, участвующие в ограничениях целостности, могут храниться в этом узле, а могут и не храниться. Важнейший учитываемый параметр – стоимость передачи данных, в т. ч. сообщений, от одного узла другому. Далее мы обсудим стратегии, различающиеся по этому критерию.

Индивидуальные ограничения

Рассматривается два случая. Если транзакция обновления – команда вставки, то все вставляемые кортежи явно предоставлены пользователем. В этом случае все индивидуальные ограничения можно проверять в узле, инициировавшем обновление. Если речь идет об обновлении с условием (команды удаления или модификации), то оно рассылается всем узлам, где хранится обновляемое отношение. Процессор запросов вычисляет условие обновления для каждого фрагмента. Результирующие кортежи в каждом узле объединяются в одно временное отношение в случае команды удаления или в два – в случае команды модификации (R^+ и R^-). Каждый узел, участвующий в распределенном обновлении, проверяет утверждения, релевантные этому узлу (например, ограничения области определения в случае удаления).

Ограничения, ориентированные на обработку множеств

Сначала изучим на примере ограничения с одним отношением. Рассмотрим функциональную зависимость из примера 3.14. Априорный тест, ассоциированный с обновлением типа **INSERT**, имеет вид

(EMP, **INSERT**, *C*),

где *C* равно

(∀e ∈ EMP)(∀NEW1 ∈ EMP)(∀NEW2 ∈ EMP) (1)
(NEW1.ENO = e.ENO ⇒ NEW1.ENAME = e.ENAME) ∧ (2)
(NEW1.ENO = NEW2.ENO ⇒ NEW1.ENAME = NEW2.ENAME (3)

Во второй строке определения *C* проверяется ограничение между вставленными (NEW1) и существующими (*e*) кортежами, а в третьей – между самими вставленными кортежами. Именно поэтому в первой строке объявлены две переменные (NEW1 и NEW2).

Теперь рассмотрим обновление EMP. Сначала процессор запросов вычисляет условие обновления и возвращает одно или два временных отношения, как в случае индивидуальных ограничений. Эти временные отношения затем рассылаются всем узлам, в которых хранится EMP. Предположим, что командой обновления является **INSERT**. Тогда каждый узел, где хранится фрагмент EMP, проверяет описанное выше ограничение *C*. Поскольку переменной *e* в *C* предшествует квантор всеобщности, *C* должно удовлетворяться для локальных данных в каждом узле, т. к. выражение $\forall x \in \{a_1, \ldots, a_n\} f(x)$ эквивалентно $[f(a_1) \wedge f(a_2) \wedge \cdots \wedge f(a_n)]$. Таким образом, узел, инициировавший обновление, должен получить от каждого узла сообщение, подтверждающее, что ограничение удовлетворяется. Если для какого-то узла это не так, то он отправляет сообщение об ошибке, говорящее, что ограничение нарушено. В таком случае обновление недопустимо, и диспетчер целостности должен решить, нужно ли отвергнуть всю транзакцию, и если да, обратиться к глобальному диспетчеру транзакций.

Далее рассмотрим ограничения с несколькими отношениями. Для простоты предположим, что в ограничения целостности входит не более одной кортежной переменной над одним отношением – это наиболее распространенный случай. Как и для ограничений с одним отношением, обновление вычисляется в инициировавшем его узле. Проверка производится в главном узле запроса с применением алгоритма ENFORCE (см. алгоритм 3.2).

Пример 3.26. Проиллюстрируем этот алгоритм на примере, основанном на ограничении внешнего ключа из примера 3.13. Пусть *u* – вставка нового кортежа в ASG. В описанном выше алгоритме используется априорный тест (ASG, **INSERT**, *C*), где *C* имеет вид

∀NEW ∈ ASG^+, ∃*j* ∈ PROJ: NEW.PNO = *j*.PNO.

Для этого ограничения команда извлечения выбирает из ASG^+ все новые кортежи, для которых *C* ложно. Это утверждение можно выразить на SQL в виде

```
SELECT NEW.*
FROM   ASG+ NEW, PROJ
WHERE  COUNT(PROJ.PNO WHERE NEW.PNO = PROJ.PNO)=0
```

Отметим, что NEW.* обозначает все атрибуты ASG+. ♦

Алгоритм 3.2. ENFORCE

Вход: *U*: тип обновления; R: отношение
begin
  извлечь все откомпилированные утверждения (R, *U*, C_i)
  inconsistent ← **false**
  for *каждого откомпилированного утверждения* **do**
    result ← все новые (соответственно старые) кортежи R, где ¬(C_i)
  end for
  if *card*(*result*) ≠ 0 **then**
    inconsistent←**true**
  end if
  if ¬*inconsistent* **then**
    разослать кортежи, подлежащие обновлению, во все узлы, где
    хранятся фрагменты R
  else
    отвергнуть обновление
  end if
end

Таким образом, стратегия заключается в том, чтобы разослать новые кортежи узлам, в которых хранится отношение PROJ, чтобы выполнить соединения, а затем собрать все результаты в главном узле запроса. Каждый узел, где хранится фрагмент PROJ, соединяет свой фрагмент с ASG+ и отправляет результат главному узлу запроса, который объединяет все результаты. Если объединение пусто, то база данных согласована. В противном случае обновление перевело бы базы данных в несогласованное состояние и потому должно быть отвергнуто средствами глобального менеджера транзакций. Можно предложить и более изощренные стратегии, которые уведомляют о несогласованности или компенсируют ее.

Ограничения с агрегатами

Эти ограничения относятся к числу самых дорогих для проверки, потому что требуют вычисления агрегатных функций. Типичные агрегатные функции – **MIN**, **MAX**, **SUM** и **COUNT**. Каждая агрегатная функция содержит две части: проекцию и выборку. Для эффективной проверки ограничений нужно порождать априорные тесты, изолирующие избыточные данные, которые можно хранить в каждом узле, где хранится ассоциированное отношение. В разделе 3.1.2 мы называли эти данные *материализованными представлениями*.

3.3.2.3. Итоги обсуждения распределенного контроля целостности

Главная проблема распределенного контроля целостности состоит в очень высоких, иногда неприемлемых затратах на сетевой обмен и обработку при проверке ограничений. Две основные проблемы проектирования менеджера распределенной целостности – определение распределенных утверждений и разработка алгоритмов проверки, которые минимизировали бы стоимость распределенной проверки ограничений. В этой главе мы показали, что распределенный контроль целостности вполне достижим, если обобщить превентивный метод, основанный на компиляции ограничений семантической целостности в априорные тесты. Метод является общим, поскольку можно обработать все типы ограничений, допускающие выражение в логике предикатов первого порядка. Он совместим с определением фрагментов и минимизирует объем межузловых коммуникаций. Повысить производительность распределенной проверки целостности можно, если тщательно определять фрагменты. Поэтому спецификация распределенных ограничений целостности – важный аспект процесса проектирования распределенной базы данных.

В описанном выше методе предполагается наличие глобальной поддержки транзакций. Если такой поддержки нет, как в некоторых слабо связанных мультибазовых системах, то проблема осложняется. Во-первых, интерфейс между диспетчером ограничений и компонентной СУБД другой, потому что проверка ограничений больше не является частью глобального механизма валидации транзакций. Вместо этого компонентная СУБД должна уведомлять диспетчер целостности о необходимости проверить ограничения после некоторых событий, например в процессе фиксации локальных транзакций. Это можно сделать с помощью триггеров, в которых событием является обновление отношений, встречающихся в глобальных ограничениях. Во-вторых, если обнаружено нарушение глобального ограничения, то поскольку не существует глобальной отмены, следует предусмотреть специальные корректирующие транзакции, которые переводят базу данных в глобально согласованное состояние. Решением может стать семейство протоколов для проверки глобальной целостности. Корневой будет простая стратегия, основанная на вычислении дифференциальных отношений (как в описанном выше методе), которая гарантированно безопасна (правильно идентифицирует нарушения ограничений), но, возможно, неточна (может генерировать событие ошибки, даже если никакие ограничения не нарушены). Неточность связана с тем, что порождение дифференциальных отношений в разные моменты времени в разных узлах может давать *фантомные*, т. е. никогда не существовавшие, состояния глобальной базы данных. Для решения этой проблемы предложено дополнить базовый протокол временными метками или использовать локальные команды транзакций.

3.4. Заключение

Контроль данных включает управление представлениями, контроль доступа и контроль семантической целостности. В реляционных СУБД эти функции можно реализовать единообразно, включив правила, описывающие контроль над манипуляциями данными. Решения, первоначально предназначенные для реализации этих функций в централизованных системах, были значительно расширены и дополнены для распределенных систем. В частности, это относится к поддержке материализованных представлений и к избирательному контролю доступа на основе групп. Контролю семантической целостности уделялось меньше внимания, и, вообще говоря, он не слишком хорошо поддерживается в распределенных СУБД.

В целом контроль данных в распределенных системах более сложен и обходится дороже. Две главные проблемы эффективного контроля данных – определение и хранение правил (выбор узла) и проектирование алгоритмов проверки ограничений, которые минимизировали бы стоимость передачи данных по сети. Это трудная задача, потому что расширенная функциональность (и общность) ведет к увеличению объема данных, циркулирующих между узлами. Она упрощается, если правила контроля полностью реплицируются на все узлы, и усложняется, если требуется сохранить автономность узлов. Возможны и дополнительные специальные оптимизации с целью минимизации затрат на контроль данных, но за это приходится расплачиваться новыми накладными расходами, например на управление материализованными представлениями или хранение избыточных данных. Таким образом, спецификацию контроля распределенных данных следует включать в проект распределенной базы данных, чтобы учесть и стоимость контроля над программами обновления.

3.5. Библиографические замечания

Тема контроля данных в централизованных системах хорошо изучена, и во всех основных СУБД имеется развитая поддержка. Исследования по контролю данных в распределенных системах начались в середине 1980-х годов в проекте R* научно-исследовательского подразделения компании IBM (IBM Research), а затем были существенно расширены в связи с появлением новых важных приложений, в т. ч. хранилищ данных и интеграции данных.

В большинстве работ по управлению представлениями рассматриваются обновление через представление и поддержка материализованных представлений. Две основополагающие работы по централизованному управлению представлениями – [Chamberlin et al. 1975] и [Stonebraker 1975]. В первой из них представлено интегрированное решение для управления представле-

ниями и авторизацией в проекте System R подразделения IBM Research. Во второй описан метод модификации запросов в проекте INGRES Калифорнийского университета в Беркли, где единообразно обрабатываются представления, авторизация и контроль семантической целостности. Этот метод был описан в разделе 3.1.

Теоретические решения проблемы обновления представлений приведены в работах [Bancilhon and Spyratos 1981, Dayal and Bernstein 1978, Keller 1982]. В пионерской работе по семантике обновления представлений [Bancilhon and Spyratos 1981] авторы формализуют свойство инвариантности представлений после обновления и показывают, как можно обновлять широкий класс представлений, включающий и соединения. Семантическая информация о базовых отношениях особенно полезна для нахождения однозначно определенного распространения обновлений. Однако в современных коммерческих системах поддержка обновления представлений сильно ограничена.

Материализованные представления стали предметом пристального внимания в контексте хранилищ данных. Понятие моментального снимка для оптимизации выведения представления в распределенных системах баз данных введено в работе [Adiba and Lindsay 1980] и обобщено в работе [Adiba 1981] как унифицированный механизм управления представлениями и снимками, а также фрагментированными и реплицированными данными. Самообслуживаемость материализованных представлений относительно различных видов обновления (вставка, удаление, модификация) рассматривается в работе [Gupta et al. 1996]. Подробное изложение управления материализованными представлениями имеется в статье [Gupta and Mumick 1999], где описаны основные методы инкрементного обслуживания материализованных представлений. Алгоритм подсчета, описанный в разделе 3.1.3, впервые предложен в работе [Gupta et al. 1993]. Мы рассказали о двух важных оптимизациях, предложенных сравнительно недавно и позволяющих заметно уменьшить время обслуживания в алгоритме подсчета, основываясь на формализме обобщенных отношений-мультимножеств [Koch 2010]. Первая оптимизация – материализовать представления, соответствующие подзапросам исходного запроса [Koch et al. 2014, Berkholz et al. 2017, Nikolic and Olteanu 2018], вторая – использовать асимметрию данных [Kara et al. 2019].

Безопасность в компьютерных системах вообще рассмотрена в книге [Hoffman 1977]. Безопасность в централизованных системах баз данных – тема работы [Lunt and Fernández 1990, Castano et al. 1995]. Избирательный контроль доступа (DAC) в распределенных системах впервые стал предметом рассмотрения в контексте проекта R*. Механизм контроля доступа, реализованный в System R [Griffiths and Wade 1976], обобщен в работе [Wilms and Lindsay 1981] на группы пользователей и на работу в распределенной среде. Мандатный контроль доступа (MAC) для распределенных СУБД привлек значительный интерес недавно. Основополагающая работа по MAC – статья [Bell and LaPadula 1976], в которой разработана модель Белла–Лападулы. MAC для баз данных описан в работе [Lunt and Fernández 1990, Jajodia and Sandhu 1991]. Хорошее введение в многоуровневую безопасность в реляционных СУБД – работа [Rjaibi 2004]. Управление транзакциями в многоуровневой безопасной СУБД рассматривается в работах [Ray et al. 2000, Jajodia

et al. 2001]. Обобщение MAC на распределенные СУБД предложено в работе [Thuraisingham 2001]. Контроль доступа на основе ролей (RBAC) [Ferraiolo and Kuhn 1992] обобщает DAC и MAC путем введения ролей как механизма, обеспечивающего независимость субъектов и объектов. Базы данных Гиппократа [Sandhu et al. 1996] ассоциируют с данными информацию о целях, т. е. причины сбора данных и доступа к ним.

Материал раздела 3.3 взят в основном из работ по контролю семантической целостности [Simon and Valduriez 1984, 1986, 1987]. В частности, в статье [Simon and Valduriez 1986] превентивная стратегия централизованного контроля целостности, основанная на априорных тестах, обобщена на работу в распределенной среде при наличии глобальной поддержки транзакций. Оригинальная идея декларативных методов, т. е. использования утверждений логики предикатов для задания ограничений целостности, принадлежит Флорентину [Florentin 1974]. Наиболее важные декларативные методы описаны в работах [Bernstein et al. 1980a, Blaustein 1981, Nicolas 1982, Simon and Valduriez 1984, Stonebraker 1975]. Понятие конкретных представлений для хранения избыточных данных описано в работе [Bernstein and Blaustein 1982]. Заметим, что конкретные представления полезны для оптимизации проверки ограничений с агрегатами. В работах [Civelek et al. 1988, Sheth et al. 1988b и Sheth et al. 1988a] описаны системы и инструменты для контроля данных, в особенности для управления представлениями. Проверка семантической целостности в слабо связанных многобазовых системах без глобальной поддержки транзакций рассматривается в работе [Grefen and Widom 1997].

Упражнения

Задача 3.1. Пользуясь SQL-подобным синтаксисом, определите представление `V(ENO, ENAME, PNO, RESP)`, где продолжительность работы над проектом равна 24. Является ли `V` обновляемым? Предположим, что отношения `EMP` и `ASG` горизонтально фрагментированы на основе частот доступа следующим образом:

Узел 1	Узел 2	Узел 3
EMP_1	EMP_2	
	ASG_1	ASG_2

где

$EMP_1 = \sigma_{TITLE \neq "Engineer"}(EMP)$
$EMP_2 = \sigma_{TITLE = "Engineer"}(EMP)$
$ASG_1 = \sigma_{0<DUR<36}(ASG)$
$ASG_2 = \sigma_{DUR \geq 36}(ASG)$

В каком узле (или узлах) нужно сохранить определение `V` без полной репликации, чтобы повысить локальность ссылок?

Задача 3.2. Выразите следующий запрос: получить из представления V имена работников, трудящихся над проектом CAD/CAM.

Задача 3.3 (*). Предположим, что отношение PROJ горизонтально фрагментировано следующим образом:

$$PROJ_1 = \sigma_{PNAME="CAD/CAM"}(PROJ)$$
$$PROJ_2 = \sigma_{PNAME\neq"CAD/CAM"}(PROJ)$$

Модифицируйте запрос из задачи 3.2, преобразовав его в запрос к фрагментам.

Задача 3.4 (**). Предложите распределенный алгоритм, который эффективно обновляет снимок в одном узле, полученный проецированием отношения, горизонтально фрагментированного на два других узла. Приведите пример запроса к представлению и базовым отношениям, который дает несогласованный результат.

Задача 3.5 (*). Рассмотрим представление EG из примера 3.5, в котором в качестве базовых данных используются отношения EMP и ASG, и предположим, что его состояние выводится из примера 3.1, так что EG содержит 9 кортежей (рис. 3.4). Предположим, что кортеж ⟨E3, P3, Consultant, 10⟩ отношения ASG обновляется и принимает вид ⟨E3, P3, Engineer, 10⟩. Примените базовый алгоритм подсчета для обновления представления EG. Какие атрибуты следует добавить в представление EG, чтобы сделать его самообслуживаемым?

Задача 3.6. Предложите схему отношения для хранения прав доступа, ассоциированных с группами пользователей в каталоге распределенной базы данных, а также схему фрагментации этого отношения в предположении, что все члены группы находятся в одном узле.

Задача 3.7 (**). Предложите алгоритм выполнения команды **REVOKE** в распределенной СУБД в предположении, что привилегия **GRANT** может быть предоставлена только группе пользователей, все члены которой находятся в одном узле.

Задача 3.8 (**). Рассмотрим многоуровневое отношение PROJ** на рис. 3.8. В предположении, что существует только два уровня секретности для атрибутов (S и C), предложите размещение PROJ** в двух узлах с использованием фрагментации и репликации, так чтобы избежать скрытых каналов в запросах чтения. Обсудите, какие ограничения нужно наложить на обновления, чтобы это размещение работало.

Задача 3.9. Пользуясь языком задания ограничений целостности, представленным в этой главе, выразите ограничение целостности, согласно которому продолжительность работы над проектом не может превышать 48 месяцев.

Задача 3.10 (*). Определите априорные тесты, ассоциированные с ограничениями целостности, рассмотренными в примерах 3.11–3.14.

Задача 3.11. Рассмотрим следующую вертикальную фрагментацию отношений EMP, ASG и PROJ:

Узел 1	Узел 2	Узел 3	Узел 4
EMP_1	EMP_2		
	$PROJ_1$	$PROJ_2$	
		ASG_1	ASG_2

где

$$EMP_1 = \prod_{ENO,ENAME}(EMP)$$
$$EMP_2 = \prod_{ENO,TITLE}(EMP)$$
$$PROJ_1 = \prod_{PNO,PNAME}(PROJ)$$
$$PROJ_2 = \prod_{PNO,BUDGET}(PROJ)$$
$$ASG_1 = \prod_{ENO,PNO,RESP}(ASG)$$
$$ASG_2 = \prod_{ENO,PNO,DUR}(ASG)$$

Где следует хранить априорные тесты, найденные при решении задачи 3.9?

Задача 3.12 (**). Рассмотрим следующее ограничение, ориентированное на обработку множеств:

```
CHECK ON e:EMP, a:ASG
  (e.ENO = a.ENO AND (e.TITLE = "Programmer")
  IF a.RESP = "Programmer")
```

Что оно означает? В предположении, что EMP и ASG размещены, как в предыдущем упражнении, определите соответствующие априорные тесты и способы их хранения. Примените алгоритм ENFORCE к обновлению ASG типа **INSERT**.

Задача 3.13 (**). Предположим, что имеется распределенная мультибазовая система без глобальной поддержки транзакций. Предположим также, что имеется два узла, в каждом из которых хранится свое отношение EMP (они разные), и диспетчер целостности, который взаимодействует с компонентной СУБД. Пусть необходимо глобальное ограничение уникального ключа для EMP. Предложите простую стратегию, в которой для проверки этого ограничения используются дифференциальные отношения. Обсудите возможные действия в случае нарушения ограничения.

Глава 4

Распределенная обработка запросов

Благодаря сокрытию низкоуровневых деталей физической организации данных языки реляционных баз данных позволяют выражать сложные запросы в простой и лаконичной форме. Для получения ответа на запрос пользователю не нужно точно описывать процедуру его обработки, это автоматически делает модуль, называемый *процессором запросов*. Тем самым пользователь освобождается от оптимизации запроса – утомительной процедуры, с которой процессор запросов справится лучше, т. к. в его распоряжении огромный объем полезной информации о данных.

Поскольку вопрос о производительности критически важен, обработка запросов привлекала (и продолжает привлекать) много внимания в контексте как централизованных, так и распределенных баз данных. Однако в распределенной среде проблема обработки запросов намного труднее из-за большого количества параметров, влияющих на производительность. В частности, отношения, участвующие в распределенном запросе, могут быть фрагментированы и (или) реплицированы, что влечет за собой затраты на передачу данных. Кроме того, при наличии большого числа узлов время получения ответа может резко возрасти.

В этой главе мы подробно расскажем об обработке запросов в распределенных СУБД. В качестве контекста мы выбрали реляционное исчисление и реляционную алгебру в силу их общности и широкого использования в распределенных СУБД. В главе 2 мы видели, что распределенные отношения реализуются с помощью фрагментов с целью увеличения локальности ссылок, а иногда еще и распараллеливания наиболее важных запросов. Роль процессора распределенных запросов заключается в отображении высокоуровневого запроса (по предположению, выраженного в терминах реляционного исчисления) к распределенной базе данных (т. е. набору глобальных отношений) в последовательность операторов базы данных (реляционной алгебры), применяемых к фрагментам отношения. У этого отображения есть несколько важных функций. Во-первых, *запрос исчисления* необходимо *разложить* в последовательность реляционных операторов, называемую *алгебраическим запросом*. Во-вторых, данные, к которым обращается запрос, должны быть *локализованы*, т. е. операторы, выраженные в терминах отношений, преобразуются в операторы, применяемые к локальным данным

(фрагментам). Наконец, алгебраический запрос к фрагментам должен быть дополнен коммуникационными операторами и *оптимизирован* относительно некоторой функции стоимости. Обычно функция стоимости выражается в терминах вычислительных ресурсов: операций дискового ввода-вывода, процессорного времени и потребления сети.

Эта глава организована следующим образом: в разделе 4.1 приводится обзор распределенной обработки запросов. В разделе 4.2 описывается локализация данных с упором на методы редукции и упрощения для следующих четырех типов фрагментации: горизонтальная, вертикальная, производная и гибридная. В разделе 4.3 обсуждается основная проблема оптимизации: порядок соединений в распределенных запросах. Исследуются также альтернативные стратегии соединения, основанные на полусоединении. В разделе 4.4 обсуждается распределенная модель стоимости. В разделе 4.5 иллюстрируется применение рассмотренных методов в трех основных подходах к оптимизации распределенных запросов: динамическом, статическом и гибридном. В разделе 4.5 рассматривается адаптивная обработка запросов.

Предполагается, что читатель знаком с базовыми понятиями обработки запросов в централизованных СУБД в объеме, изучаемом в университетских курсах по базам данных и в учебниках.

4.1. Общий обзор

В этом разделе рассматривается распределенная обработка запросов. Сначала в разделе 4.1.1 обсуждается постановка задачи об обработке запросов. Раздел 4.1.2 содержит введение в оптимизацию запросов. Наконец, в разделе 4.1.3 описываются различные уровни обработки запроса, начиная с распределенного запроса и заканчивая выполнением операторов в локальных узлах.

4.1.1. Задача обработки запроса

Основная функция процессора запросов – преобразовать высокоуровневый запрос (обычно выраженный средствами реляционного исчисления) в эквивалентный низкоуровневый запрос (обычно в терминах того или иного варианта реляционной алгебры). Низкоуровневый запрос реализует стратегию выполнения запроса. Преобразование должно обеспечивать корректность и эффективность. Оно является корректным, если низкоуровневый запрос имеет такую же семантику, как исходный, т. е. оба запроса дают одинаковый результат. Точно определенное отображение реляционного исчисления на реляционную алгебру без труда решает проблему корректности. Но выработать эффективную стратегию гораздо труднее. Для запроса в терминах реляционного исчисления может существовать много эквивалентных корректных преобразований на язык реляционной алгебры. Но эквивалентные стратегии выполнения могут совершенно по-разному потреблять ресурсы компьютера, поэтому главная трудность – выбрать такую стратегию, которая минимизирует потребление ресурсов.

Пример 4.1. Рассмотрим следующее подмножество схемы базы данных конструкторской компании:

```
EMP(ENO, ENAME, TITLE)
ASG(ENO, PNO, RESP, DUR)
```

и простой запрос:

Найти имена работников-менеджеров проектов.

Выражение этого запроса в реляционном исчислении с применением синтаксиса SQL (естественного соединения) имеет вид

```
SELECT ENAME
FROM   EMP NATURAL JOIN ASG
WHERE  RESP = "Manager"
```

Корректными преобразованиями этого запроса являются два эквивалентных выражения реляционной алгебры:

$$\Pi_{\text{ENAME}}(\sigma_{\text{RESP="Manager"} \wedge \text{EMP.ENO=ASG.ENO}}(\text{EMP} \times \text{ASG}))$$

и

$$\Pi_{\text{ENAME}}(\bowtie_{\text{ENO}}(\sigma_{\text{RESP="Manager"}}(\text{ASG}))).$$

Сразу видно, что второй запрос, в котором нет декартова произведения EMP и ASG, потребляет гораздо меньше вычислительных ресурсов, поэтому его и следует предпочесть. ♦

Для выражения стратегии вычисления в распределенной системе реляционной алгебры недостаточно. Ее нужно дополнить операторами обмена данными между узлами. Процессор распределенных запросов должен выбрать не только порядок выполнения операторов реляционной алгебры, но и лучшие узлы для обработки данных и, возможно, способ преобразования данных. Это увеличивает пространство решений, из которого выбирается стратегия распределенного выполнения, поэтому обработка распределенных запросов оказывается гораздо более трудной задачей, чем в централизованном случае.

Пример 4.2. В этом примере иллюстрируется важность выбора узлов и коммуникаций для запроса реляционной алгебры к фрагментированной базе данных. Рассмотрим следующий запрос из примера 4.1:

$$\Pi_{\text{ENAME}}(\text{EMP} \bowtie_{\text{ENO}} (\sigma_{\text{RESP="Manager"}}(\text{ASG}))).$$

Предположим, что отношения EMP и ASG горизонтально фрагментированы следующим образом:

$$\text{EMP}_1 = \sigma_{\text{ENO} \leq \text{"E3"}}(\text{EMP});$$
$$\text{EMP}_2 = \sigma_{\text{ENO} > \text{"E3"}}(\text{EMP});$$
$$\text{ASG}_1 = \sigma_{\text{ENO} \leq \text{"E3"}}(\text{ASG});$$
$$\text{ASG}_2 = \sigma_{\text{ENO} > \text{"E3"}}(\text{ASG}).$$

Фрагменты ASG_1, ASG_2, EMP_1 и EMP_2 хранятся в узлах 1, 2, 3 и 4 соответственно, а результат ожидается в узле 5.

Для простоты мы будем далее игнорировать оператор проекции. На рис. 4.1 показаны две эквивалентные стратегии распределенного выполнения этого запроса. Стрелка, ведущая из узла *i* в узел *j* и снабженная меткой R, означает, что отношение R передается из *i* в *j*. В стратегии A используется тот факт, что отношения EMP и ASG фрагментированы одинаково, чтобы можно было выполнять операторы выборки и соединения параллельно. Стратегия B, подразумеваемая по умолчанию (всегда работает), просто централизует все данные-операнды в узле результата, перед тем как обрабатывать запрос.

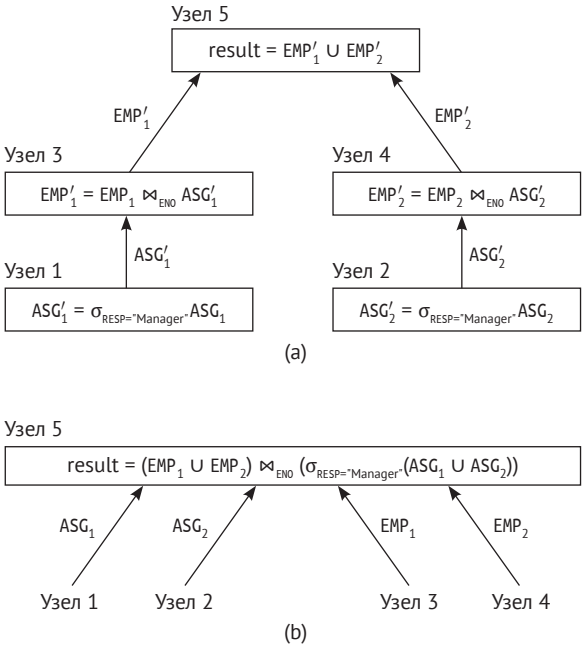


Рис. 4.1 ❖ Эквивалентные стратегии распределенного выполнения: (a) стратегия A; (b) стратегия B

Для оценки потребления ресурсов этими двумя стратегиями мы воспользуемся очень простой моделью. Предположим, что доступ к кортежу *tupacc* стоит 1 единицу (в чем он измеряется, мы не уточняем), а передача кортежа *tuptrans* – 10 единиц. Предположим, что отношения EMP и ASG содержат 400 и 1000 кортежей соответственно, причем в отношении ASG имеется 20 менеджеров. Еще предположим, что данные распределены между узлами равномерно. Наконец, будем считать, что отношения ASG и EMP локально кластеризованы по атрибутам RESP и ENO соответственно. Таким образом, имеется прямой доступ к кортежам ASG (соответственно EMP) по значению атрибута RESP (соответственно ENO).

Полную стоимость стратегии A можно вычислить следующим образом.

1.	Порождение ASG′ в результате выборки из ASG требует (10 + 10) * *tupacc*	=	20
2.	Передача ASG′ в узлы, где хранится EMP, требует (10 + 10) * *tuptrans*	=	200
3.	Порождение EMP′ путем соединения ASG и EMP требует (10 + 10) * *tupacc* * 2	=	40
4.	Передача EMP′ в узел результата требует (10 + 10) * *tuptrans*	=	200
	Полная стоимость равна		460

Стоимость стратегии B можно вычислить следующим образом.

1.	Передача EMP в узел 5 требует 400 * *tuptrans*	=	4000
2.	Передача ASG в узел 5 требует 1000 * *tuptrans*	=	10 000
3.	Порождение ASG′ в результате выборки из ASG требует 1000 * *tupacc*	=	1000
4.	Соединение EMP и ASG требует 400 * 20 * *tupacc*	=	8000
	Полная стоимость равна		23 000

В стратегии A для соединения ASG и EMP (шаг 3) можно использовать кластерный индекс по атрибуту ENO отношения EMP. Таким образом, к EMP нужно обратиться только один раз для каждого кортежа ASG. В стратегии B мы предполагаем, что методы доступа к отношениям EMP и ASG, основанные на атрибутах RESP и ENO, недоступны из-за необходимости передавать данные. На практике это разумное предположение. Мы предполагаем, что соединение EMP и ASG на шаге 4 производится принятым по умолчанию алгоритмом вложенных циклов (который просто вычисляет декартово произведение двух входных отношений). Стратегия A лучше B в 50 раз – очень много. Кроме того, она обеспечивает лучшее распределение работы между узлами. Разница была бы еще больше, если бы мы сделали предположение о низкой скорости связи и (или) более высокой степени фрагментации. ♦

4.1.2. Оптимизация запроса

Под оптимизацией запроса понимается процесс создания плана выполнения запроса (ПВЗ), который представляет стратегию выполнения запроса. ПВЗ минимизирует целевую функцию стоимости. В оптимизаторе запросов – модуле, выполняющем эту операцию, – принято выделять три компоненты: пространство поиска, модель стоимости и стратегию поиска.

4.1.2.1. Пространство поиска

Пространство поиска – это множество альтернативных планов выполнения входного запроса. Все эти планы эквивалентны в том смысле, что дают один и тот же результат, но отличаются порядком выполнения и реализацией операторов, а значит, и производительностью. Пространство поиска строится путем применения правил преобразования, например правил реляционной алгебры.

4.1.2.2. Модель стоимости

Модель стоимости служит для предсказания стоимости данного плана выполнения и для сравнения эквивалентных планов с целью выбрать из них лучший. Чтобы быть точной, модель стоимости должна хорошо знать о среде распределенного выполнения, использовании статистики данных и функциях стоимости.

В распределенной базе данных статистика обычно собирается на уровне фрагментов и включает количество фрагментов, их размеры, а также количество различных значений каждого атрибута. Для сведения к минимуму вероятности ошибки иногда используется более детальная статистика, например гистограммы значений атрибутов, но за это приходится расплачиваться увеличением затрат на управление. Точность статистики обеспечивается путем периодического обновления.

Хорошей мерой затрат может служить *полная стоимость* обработки запроса. Это суммарное время, затраченное на обработку всех операторов запроса в различных узлах и на межузловую передачу данных. Еще одной мерой затрат является *время ответа* на запрос, т. е. время, потраченное на обработку запроса. Поскольку операторы могут параллельно выполняться в разных узлах, время ответа может быть значительно меньше полной стоимости.

В распределенной системе баз данных подлежащая минимизации полная стоимость включает затраты на процессор, ввод-вывод и коммуникацию. Затраты на процессор возникают при выполнении операторов в основной памяти. Затраты на ввод-вывод – это время, потраченное на доступ к диску. Эту величину можно уменьшить, сократив количество операций доступа за счет применения более быстрых методов доступа к данным и эффективного использования основной памяти (управления буферами). Затраты на коммуникацию – это время, необходимое для обмена данными между узлами, участвующими в выполнении запроса. Они возникают при обработке сообщений (упаковке и распаковке) и передаче данных по каналам связи.

Затраты на коммуникацию – пожалуй, самый важный фактор при оценке распределенных баз данных. В большинстве ранних предложений по распределенной оптимизации запросов предполагалось, что затраты на коммуникацию доминируют над стоимостью локальной обработки (ввода-вывода и процессора), поэтому последние вообще игнорировались. Однако в современных средах распределенной обработки сети связи стали гораздо быстрее, сравнявшись по скорости с дисками. Поэтому требуется учитывать взвешенную сумму всех трех компонентов, поскольку каждый из них вносит заметный вклад в полную стоимость обработки запроса.

В этой главе мы считаем реляционную алгебру основным средством для выражения результата обработки запроса. Поэтому сложность операторов реляционной алгебры, которая прямо влияет на время их выполнения, диктует некоторые принципы, полезные при создании процессора запросов. Эти принципы помогают при выборе окончательной стратегии выполнения.

Проще всего определить сложность в терминах мощности отношений вне зависимости от таких физических деталей реализации, как фрагментация и структура хранения. Сложность унарных операторов равна $O(n)$, где n –

мощность отношения, если результирующие кортежи можно получить независимо друг от друга. Сложность бинарных операторов равна $O(n \log n)$, если каждый кортеж одного отношения необходимо сравнивать с каждым кортежем другого на основе равенства избранных атрибутов. Эта оценка сложности предполагает, что кортежи одного отношения отсортированы по сравниваемым атрибутам. Но при использовании хеширования и наличии достаточной памяти для хранения одного хешированного отношения целиком сложность бинарных операторов можно сократить для $O(n)$. Для операторов проецирования с устранением дубликатов и группировки необходимо сравнивать кортежи отношения каждый с каждым, поэтому сложность также составляет $O(n \log n)$. Наконец, сложность вычисления декартова произведения двух отношений составляет $O(n^2)$, потому что каждый кортеж одного отношения нужно объединить с каждым кортежем другого.

4.1.2.3. Стратегия поиска

Стратегия поиска предназначена для исследования пространства поиска и выбора наилучшего плана согласно модели стоимости. Она определяет, какие планы просматривать и в каком порядке. Особенности распределенной среды отражаются в пространстве поиска и в модели стоимости.

Прямой метод оптимизации запроса заключается в том, чтобы организовать исчерпывающий поиск в пространстве решений, предсказать стоимость каждой стратегии и выбрать стратегию с минимальной стоимостью. Конечно, этот метод найдет наилучшую стратегию, но затраты на саму оптимизацию могут быть очень значительны. Проблема в том, что пространство решений может быть большим, т. е. количество эквивалентных стратегий велико, даже если в запросе не так уж много отношений. Все становится еще хуже, когда количество отношений или фрагментов возрастает (например, становится больше 10). Впрочем, высокие затраты на оптимизацию не всегда плохи, особенно если запрос оптимизируется один раз, а исполняется многократно.

Самая популярная стратегия поиска, применяемая оптимизаторами запросом, – *динамическое программирование*. Впервые она была предложена в проекте System R подразделения IBM Research. *Построение* планов начинается с базовых отношений, на каждом шаге производится соединение с одним отношением, пока не будут получены полные планы. В процессе динамического программирования строятся все возможные планы, в порядке просмотра «в ширину», и только потом выбирается «лучший». Чтобы уменьшить стоимость оптимизации, частичные планы, которые вряд ли приведут к оптимальному, как можно раньше *отсекаются* (т. е. отбрасываются).

Для очень сложных запросов, когда пространство поиска чрезмерно велико, можно использовать *рандомизированные* стратегии, например итеративное улучшение и имитацию отжига. Они пытаются найти хорошее решение; необязательно оно окажется лучшим, зато будет соблюден компромисс между временем оптимизации и временем выполнения.

Еще один подход – с другого боку – ограничить пространство решений, так чтобы нужно было рассматривать не очень много стратегий. И в централизованных, и в распределенных системах распространенной эвристикой

является минимизация размера промежуточных отношений. Для этого можно сначала выполнять унарные операторы, а бинарные упорядочивать по возрастанию размера промежуточных отношений.

Запрос можно оптимизировать в разные моменты времени относительно момента его выполнения. Можно оптимизировать *статически* – до выполнения запроса, или *динамически* – во время выполнения запроса. Статическая оптимизация производится на этапе компиляции запроса, поэтому ее стоимость амортизируется между многократными выполнениями запроса. Следовательно, такой подход выгоден при использовании исчерпывающего поиска. Поскольку размеры промежуточных отношений неизвестны до момента выполнения, остается только оценить их с использованием статистики базы данных. Ошибки на этапе оценки могут привести к выбору неоптимальной стратегии.

Динамическая оптимизация производится во время выполнения запроса. В любой момент выполнения выбор следующего оптимального оператора может основываться на точном знании результатов ранее выполненных операторов. Поэтому для оценки размера промежуточных результатов статистика базы данных не нужна. Однако она все же может пригодиться для выбора первых операторов. Основное преимущество динамической оптимизации над статической в том, что процессору запросов доступны истинные размеры промежуточных отношений, так что вероятность неправильного выбора сводится к минимуму. А основной недостаток в том, что оптимизацию запросов – дорогостоящую операцию – нужно повторять при каждом выполнении запроса. Поэтому динамическая оптимизация больше подходит для ситуативных запросов, выполняемых однократно.

Гибридная оптимизация запросов – попытка сохранить плюсы статической оптимизации, избежав проблем, связанных с неточной оценкой. В основном этот подход статический, но если обнаружена существенная разница между предсказанным и фактическим размером промежуточных отношений, то может применяться динамическая оптимизация на этапе выполнения.

4.1.3. Уровни обработки запросов

Задачу обработки запросов можно разложить на несколько подзадач, соответствующих различным слоям. На рис. 4.2 показана схема уровней обработки запросов, на которой каждый уровень решает четко определенную подзадачу. Чтобы упростить обсуждение, предположим, что статический процессор запросов не использует реплицированные фрагменты. Входом является запрос к глобальным данным, выраженный в терминах реляционного исчисления. Этот запрос содержит только имена глобальных (распределенных) отношений, т. е. распределение данных скрыто. Можно выделить четыре основных уровня распределенной обработки запросов. Первые три отображают входной запрос на распределенный план выполнения запроса (распределенный ПВЗ). Это *декомпозиция запроса*, *локализация данных* и *глобальная оптимизация запроса*. Декомпозиция запроса и локализация данных соответствуют переписыванию запроса. Первые три уровня выполняются

в центральном управляющем узле с использованием схемы, хранящейся в глобальном каталоге. Четвертый уровень отвечает за *распределенное выполнение запроса*, он исполняет план и возвращает ответ на запрос. Это делается в локальных узлах и в управляющем узле. Далее в этом разделе мы рассмотрим эти четыре уровня.

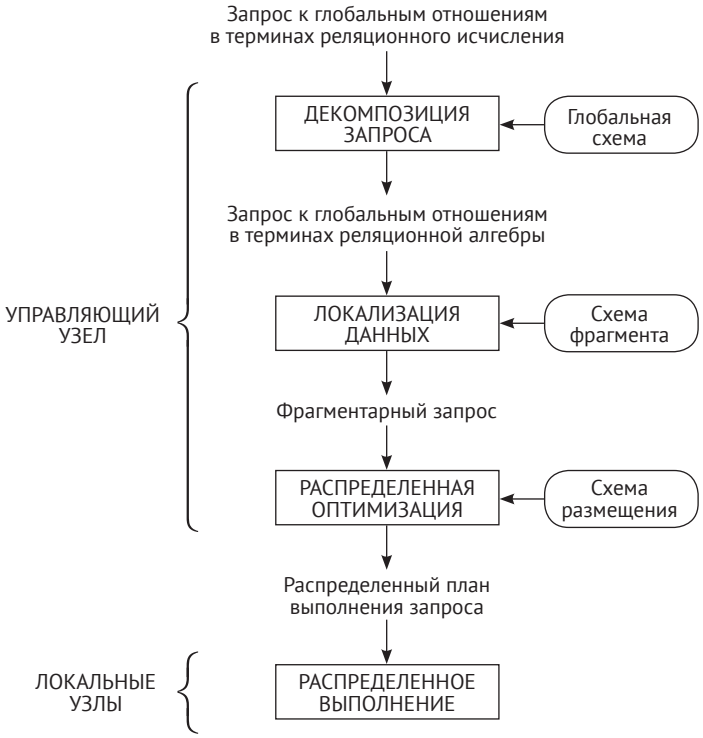


Рис. 4.2 ❖ Общая схема разбиения распределенной обработки запроса на уровни

4.1.3.1. Декомпозиция запроса

На первом уровне производится декомпозиция запроса, выраженного в терминах реляционного исчисления, в алгебраический запрос к глобальным отношениям. Информация, необходимая для такого преобразования, хранится в глобальной концептуальной схеме, описывающей глобальные отношения. Однако информация о распределении данных используется не здесь, а на следующем уровне. Методы, применяемые на этом уровне, такие же, как в централизованных СУБД, поэтому мы лишь кратко напомним их.

Декомпозицию запроса можно рассматривать как четыре последовательных шага. Во-первых, запрос переписывается в *нормализованной* форме, удобной для последующих действий. Нормализация обычно сводится к манипулированию кванторами и условиями запроса с учетом приоритета логических операторов.

Во-вторых, нормализованный запрос подвергается семантическому анализу, в ходе которого некорректные запросы выявляются и отвергаются как можно раньше. Запрос является семантически некорректным, если его компоненты не дают никакого вклада в генерирование результата. В контексте реляционного исчисления установить семантическую корректность запросов общего вида невозможно. Однако это можно сделать для широкого класса реляционных запросов, не содержащих дизъюнкции и отрицания. Механизм основан на представлении запроса в виде графа, называется *графом запроса*, или *графом связей*. Мы определим такой граф для наиболее полезных видов запросов, включающих операторы выборки, проекции и соединения. В графе запросов одна вершина содержит результирующее отношение, а все остальные – отношения-операнды. Ребро между двумя вершинами, не являющимися результатом, представляет соединение, а ребро, конечная вершина которого является результатом, представляет проекцию. Кроме того, нерезультирующая вершина может быть помечена предикатом выборки или самосоединения (соединения отношения с собой). В графе запроса можно выделить важный подграф – *граф соединений*, в котором рассматриваются только соединения.

В-третьих, корректный запрос (все еще выраженный в терминах реляционного исчисления) *упрощается*. Один из путей упрощения запроса – исключение избыточных предикатов. Заметим, что избыточные предикаты чаще всего появляются, когда запрос является результатом преобразований, примененных системой к пользовательскому запросу. В главе 3 мы видели, что такие преобразования используются при распределенном контроле данных (контроль представлений, доступа и семантической целостности).

В-четвертых, запрос в терминах реляционного исчисления *реструктурируется* и представляется в виде алгебраического запроса. Напомним, что один и тот же запрос в терминах реляционного исчисления можно преобразовать в разные алгебраические запросы и что одни алгебраические запросы «лучше» других. Качество алгебраического запроса определяется в терминах ожидаемой производительности. Традиционный способ выполнить преобразование в «лучшую» алгебраическую форму заключается в том, чтобы начать с исходного алгебраического запроса и преобразовывать его для нахождения «хорошего». Исходный алгебраический запрос выводится непосредственно из запроса на языке реляционного исчисления путем трансляции предикатов и целевой команды в реляционные операторы в порядке просмотра запроса. Этот алгебраический запрос, являющийся результатом прямой трансляции, реструктурируется посредством применения правил преобразования. Алгебраический запрос, сгенерированный на этом уровне, хорош в том смысле, что плохих путей выполнения обычно удается избежать. Например, к отношению будет только одно обращение, даже при наличии нескольких предикатов выборки. Однако данный запрос, как правило, далек от оптимального способа выполнения, поскольку на этом уровне информация о распределении данных и размещении фрагментов не используется.

4.1.3.2. Локализация данных

Входом для второго уровня является алгебраический запрос к глобальным отношениям. Основная роль второго уровня – локализовать данные запро-

са, используя информацию о распределении данных в схеме фрагмента. В главе 2 мы видели, что отношения фрагментированы и хранятся в виде дизъюнктных подмножеств, называемых фрагментами, каждое в своем узле. На этом уровне определяется, какие фрагменты участвуют в запросе, и распределенный запрос преобразуется в запрос к фрагментам. Фрагментация определяется с помощью правил, которые можно выразить как реляционные операторы. Глобальное отношение можно реконструировать, применяя правила фрагментации, а затем вывести так называемую *программу материализации*, состоящую из операторов реляционной алгебры, которая потом воздействует на фрагменты. Локализация включает два шага. На первом запрос отображается в запрос к фрагментам путем замены каждого отношения его программой материализации. На втором фрагментарный запрос упрощается и реструктурируется, в результате чего создается другой «хороший» запрос. Упрощение и реструктуризацию можно произвести по тем же правилам, что на уровне декомпозиции. Как и на уровне декомпозиции, окончательный фрагментарный запрос обычно далек от оптимального, поскольку информация о фрагментах не используется.

4.1.3.3. Распределенная оптимизация

На вход третьего уровня подается алгебраический фрагментарный запрос. Цель оптимизации запроса – найти стратегию его выполнения, близкую к оптимальной. Стратегию выполнения распределенного запроса можно описать с помощью операторов реляционной алгебры и *коммуникационных примитивов* (операторов отправки и приема) для передачи данных между узлами. На предыдущих уровнях запрос уже был частично оптимизирован, например были удалены избыточные выражения. Но эта оптимизация не зависела от таких характеристик фрагментов, как их размещение и мощность. Кроме того, еще не определены коммуникационные операторы. Изменяя порядок операторов внутри фрагментарного запроса, можно найти много эквивалентных запросов.

Оптимизация запроса заключается в нахождении «лучшего» порядка операторов, включая коммуникационные, т. е. доставляющего минимум функции стоимости. Функция стоимости, часто определяемая в терминах единиц времени, характеризует потребление вычислительных ресурсов, например диска, процессора и сети. В общем случае это взвешенная сумма затрат на ввод-вывод, процессорную обработку и коммуникацию. Чтобы выбрать порядок операторов, необходимо предсказать стоимость выполнения при других возможных порядках. Вычисление стоимости выполнения до самого выполнения запроса (т. е. статическая оптимизация) основано на статистике фрагментов и формулах оценки мощности результатов реляционных операторов. Таким образом, решения по оптимизации зависят от размещения фрагментов и доступной статистики фрагментов, которая хранится в схеме размещения.

Важный аспект оптимизации запроса – *порядок соединений*, т. к. перестановка операций соединения в запросе может улучшить производительность на несколько порядков. Результатом уровня оптимизации запроса является оптимизированный алгебраический запрос с включенными коммуникаци-

онными операторами. Обычно он представляется и сохраняется (для последующего выполнения) в форме распределенного ПВЗ.

4.1.3.4. Распределенное выполнение

Последний уровень выполняется всеми узлами, на которых размещены участвующие в запросе фрагменты. Каждый подзапрос, выполняемый в одном узле и называемый *локальным запросом*, оптимизируется с помощью локальной схемы узла и выполняется. На этом этапе можно выбрать алгоритмы, реализующие реляционные операторы. При локальной оптимизации используются алгоритмы, применяемые в централизованных системах.

Классическая реализация реляционных операторов в системах баз данных основана на модели итераторов, которая обеспечивает конвейерный параллелизм внутри деревьев операторов. Это простая модель вытягивания, в которой операторы выполняются, начиная с корневого узла (оператор, порождающий результат) до листовых узлов (в которых осуществляется доступ к базовым отношениям). Таким образом, промежуточные результаты операторов не нужно материализовывать, поскольку кортежи порождаются по требованию и могут потребляться последующими операторами. Однако необходимо, чтобы операторы были реализованы в конвейерном режиме, в соответствии с интерфейсом открыть-следующий-закрыть. Каждый оператор должен быть реализован как итератор с тремя функциями:

1) `Open()`: инициализирует внутреннее состояние оператора, например создает хеш-таблицу;
2) `Next()`: порождает и возвращает следующий кортеж или null;
3) `Close()`: освобождает выделенные ресурсы после обработки всех кортежей.

Таким образом, итератор реализует часть, связанную с обходом в цикле while: инициализацию, инкремент, условие завершения цикла и финальную очистку. Исполнение ПВЗ производится в следующем порядке. Сначала исполнение инициализируется посредством вызова `Open()` для корня дерева операторов, после чего вызов распространяется на весь план с помощью самих операторов. Затем корневой оператор итеративно порождает следующую запись результата, распространяя вызов `Next()` по дереву операторов. Исполнение завершается, когда последний вызов `Open()` возвращает «конец» корневому оператору.

Чтобы проиллюстрировать реализацию реляционного оператора с помощью интерфейса открыть-следующий-закрыть, рассмотрим оператор соединения методом вложенных циклов, который выполняет соединение $R \bowtie S$ по атрибуту `A`. Функции `Open()` и `Next()` определены следующим образом:

```
Function Open()
    R.Open() ;
    S.Open() ;
    r := R.Next() ;

Function Next()
    while (r _= null) do
```

```
        (while (s:=S.Next()) _= null) do
            if r.A=s.A then return(r,s);
        S.close() ;
        S.open() ;
        r:=R.next() ; )
    return null;
```

Не всегда возможно реализовать оператор в конвейерном режиме. Речь идет о *блокирующих* операторах, которые должны материализовать свои входные данные в памяти или на диске, перед тем как смогут что-то вернуть. Примерами блокирующих операторов являются сортировка и соединение хешированием. Если данные уже отсортированы, то соединение слиянием, группировку и устранение дубликатов можно реализовать в конвейерном режиме.

4.2. Локализация данных

Локализация данных транслирует алгебраический запрос к глобальным отношениям во фрагментарный запрос, пользуясь информацией, хранящейся в схеме фрагмента. Наивный способ достичь этой цели – сгенерировать запрос, в котором каждое глобальное отношение заменено его программой материализации. Это можно рассматривать как замену листьев дерева операторов распределенного запроса поддеревьями, соответствующими программам материализации. В общем случае этот подход неэффективен, потому что фрагментарный запрос можно подвергнуть важным реструктуризациям и упрощениям. Далее в этой главе мы для каждого типа фрагментации представим *метод редукции*, порождающий более простые и оптимизированные запросы. Мы будем использовать различные правила преобразования и эвристики, например проталкивание унарных операторов вниз по дереву.

4.2.1. Редукция для главной горизонтальной фрагментации

Функция горизонтальной фрагментации распределяет отношение на основе предиката выборки. В обсуждении ниже мы будем использовать следующий пример.

Пример 4.3. Отношение `EMP(ENO, ENAME, TITLE)` можно расщепить на три горизонтальных фрагмента EMP_1, EMP_2 и EMP_3, определенных следующим образом:

$EMP_1 = \sigma_{ENO \leq "E3"}(EMP)$;
$EMP_2 = \sigma_{"E3" < ENO \leq "E6"}(EMP)$;
$EMP_3 = \sigma_{ENO > "E6"}(EMP)$.

Программа материализации для горизонтально фрагментированного отношения представляет собой объединение фрагментов. В нашем примере имеем:

$\text{EMP} = \text{EMP}_1 \cup \text{EMP}_2 \cup \text{EMP}_3$.

Таким образом, материализованная форма любого запроса над EMP получается его заменой на $(\text{EMP}_1 \cup \text{EMP}_2 \cup \text{EMP}_3)$. ♦

Редукция запросов к горизонтально фрагментированным отношениям в основном состоит в том, чтобы после реструктуризации поддеревьев найти среди них те, которые порождают пустые отношения, и удалить их. Горизонтальную фрагментацию можно использовать для упрощения операторов выборки и проекции.

4.2.1.1. Редукция с помощью выборки

Выборка из фрагментов, для которых условие запроса противоречит условию в правиле фрагментации, порождает пустые отношения. Если дано отношение R, горизонтально фрагментированное в виде $\text{R}_1, \text{R}_2, \ldots, \text{R}_w$, где $\text{R}_j = \sigma_{pj}(\text{R})$, то это правило можно формально записать следующим образом:

Правило 1. $\sigma_{pi}(\text{R}_j) = \emptyset$, если $\forall x$ в R : $\neg(p_i(x) \wedge p_j(x))$,
где p_i и p_j – предикаты выборки, x обозначает кортеж, а $p(x)$ – «предикат p удовлетворяется для x».

Например, предикат выборки ENO="E1" конфликтует с предикатами фрагментов EMP_2 и EMP_3 из примера 4.3 (т. е. ни один кортеж в EMP_2 или EMP_3 не может удовлетворять этому предикату). Для выявления противоречивых предикатов в случае, когда предикаты достаточно общие, нужно применять методы доказывания теорем. Однако СУБД обычно упрощают сравнение предикатов, т. к. для определения правил фрагментации (это делает администратор базы данных) поддерживают только простые предикаты.

Пример 4.4. Проиллюстрируем редукцию для горизонтальной фрагментации на примере следующего запроса:

```
SELECT *
FROM   EMP
WHERE  ENO = "E5"
```

Применение наивного подхода к локализации EMP с применением EMP_1, EMP_2 и EMP_3 дает фрагментарный запрос, показанный на рис. 4.3a. Переставив выборку с оператором объединения, легко понять, что предикат выборки противоречит предикатам EMP_1 и EMP_3, так что порождаемые отношения пусты. Редуцированный запрос просто применяется к EMP_2, как показано на рис. 4.3b. ♦

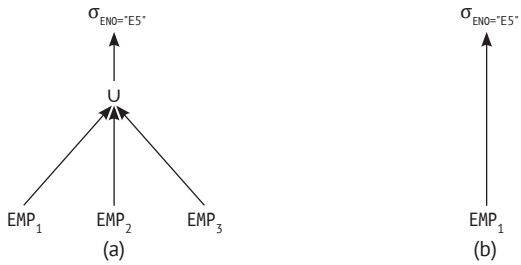


Рис. 4.3 ❖ Редукция для горизонтальной фрагментации (с помощью выборки): (a) фрагментарный запрос; (b) редуцированный запрос

4.2.2. Редукция с помощью соединения

Соединения с горизонтально фрагментированными отношениями можно упростить, если соединяемые отношения фрагментированы по атрибуту соединения. Упрощение состоит в том, чтобы заменить соединение с объединением на объединение соединений и исключить бесполезные соединения. Эта операция формально записывается в виде:

$$(R_1 \cup R_2) \bowtie S = (R_1 \bowtie S) \cup (R_2 \bowtie S),$$

где R_i – фрагменты R, а S – отношение.

После такого преобразования объединения можно будет поднять вверх в дереве операторов, так что все возможные соединения фрагментов будут хорошо видны. Бесполезные соединения фрагментов возникают, когда условия фрагментации противоречивы, так что получается пустой результат. В предположении, что фрагменты R_i и R_j определены соответственно предикатами p_i и p_j над одним и тем же атрибутом, правило упрощения можно сформулировать следующим образом:

Правило 2. $R_i \bowtie R_j = \emptyset$, если $\forall x$ в R_i $\forall y$ в R_j : $\neg(p_i(x) \wedge p_j(y))$.

Таким образом, определение и исключение бесполезных соединений по правилу 2 можно произвести, глядя только на предикаты фрагментов. Применение этого правила позволяет реализовать соединение двух отношений как параллельное частичное соединение фрагментов. Не всегда редуцированный запрос лучше (т. е. проще) фрагментарного. Фрагментарный запрос лучше, когда в редуцированном запросе много частичных соединений. Это случается, когда имеется несколько противоречивых предикатов фрагментации. Худший случай – когда каждый фрагмент одного отношения необходимо соединить с каждым фрагментом другого. Это равносильно декартову произведению двух наборов фрагментов таких, что каждый набор соответствует одному отношению. Редуцированный запрос лучше, когда число частичных соединений мало. Например, если оба отношения фрагментированы с помощью одних и тех же предикатов, то количество частичных соединений равно количеству фрагментов каждого отношения. Одно из преимуществ редуцированного запроса заключается в том, что частичные соединения можно выполнять параллельно и тем самым уменьшить время ожидания ответа.

Пример 4.5. Предположим, что отношение EMP разделено на фрагменты EMP_1, EMP_1, EMP_3, как показано выше, и что отношение ASG фрагментировано следующим образом:

$ASG_1 = \sigma_{ENO \leq "E3"}(ASG)$;
$ASG_2 = \sigma_{ENO > "E3"}(ASG)$.

В определениях фрагментов EMP_1 и ASG_1 участвует один и тот же предикат. Кроме того, предикат, определяющий ASG_2, является объединением предикатов, определяющих EMP_2 и EMP_3. Теперь рассмотрим запрос с соединением

```
SELECT *
FROM   EMP NATURAL JOIN ASG
```

Эквивалентный фрагментарный запрос показан на рис. 4.4a. Запрос, редуцированный путем замены соединения с объединением на объединение соединений и применения правила 2, можно реализовать как объединение трех частичных соединений, выполняемое параллельно (рис. 4.4b). ♦

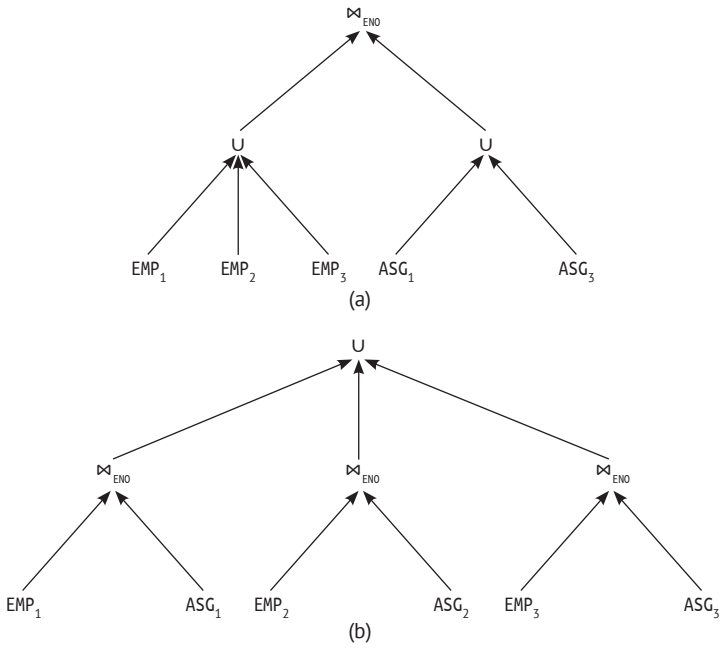


Рис. 4.4 ❖ Редукция для горизонтальной фрагментации (с помощью соединения): (a) фрагментарный запрос; (b) редуцированный запрос

4.2.3. Редукция для вертикальной фрагментации

Функция вертикальной фрагментации распределяет отношение на основе атрибутов проекции. Поскольку оператор реконструкции для вертикальной фрагментации – это соединение, то программа материализации для верти-

кально фрагментированного отношения состоит из соединения фрагментов по общему атрибуту. Вертикальную фрагментацию мы проиллюстрируем на следующем примере.

Пример 4.6. Отношение EMP можно разделить на два вертикальных фрагмента, в которых ключевой атрибут ENO дублируется:

$$EMP_1 = \prod_{ENO,ENAME}(EMP);$$
$$EMP_2 = \prod_{ENO,TITLE}(EMP).$$

Программа материализации имеет вид

$$EMP = EMP_1 \bowtie_{ENO} EMP_2.$$ ♦

Как и в случае горизонтальной фрагментации, запросы к вертикальным фрагментам можно редуцировать, выявив бесполезные промежуточные отношения и удалив порождающие их поддеревья. Проекции на вертикальный фрагмент, не имеющий общих атрибутов с проецируемыми атрибутами (кроме ключа отношения), порождают бесполезные, хотя и не пустые отношения. Пусть отношение R, определенное над атрибутами $A = \{A_1, \ldots, A_n\}$, вертикально фрагментировано в виде $R_i = \prod_{A'}(R)$, где $A' \subseteq A$. Тогда можно сформулировать следующее

Правило 3. $\prod_{D,K}(R_i)$ бесполезна, если множество атрибутов проекции D не пересекается с A'.

Пример 4.7. Проиллюстрируем применение этого правила на следующем примере:

```
SELECT ENAME
FROM   EMP
```

Эквивалентный фрагментарный запрос EMP_1 и EMP_2 (полученный, как в примере 4.4) показан на рис. 4.5a. Переставив операторы проекции и соединения (т. е. спроецировав на ENO, ENAME), легко видеть, что проекция на EMP_2 бесполезна, потому что ENAME не принадлежит EMP_2. Следовательно, проекцию нужно применить только к EMP_1, как показано на рис. 4.5b. ♦

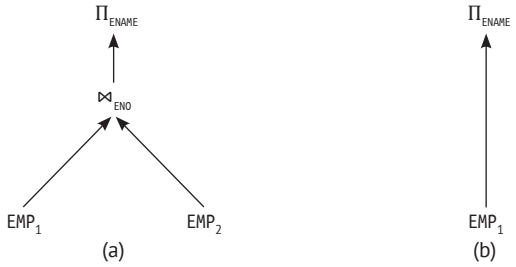


Рис. 4.5 ❖ Редукция для вертикальной фрагментации: (a) фрагментарный запрос; (b) редуцированный запрос

4.2.4. Редукция для производной фрагментации

В предыдущих разделах мы видели, что оператор соединения – быть может, самый важный из всех, потому он часто встречается и дорого обходится, – можно оптимизировать, применив главную горизонтальную фрагментацию, если соединяемые отношения фрагментированы согласованно с атрибутами, по которым производится соединение. В этом случае соединение двух отношений реализуется как объединение частичных соединений. Однако этот метод не годится, если одно из отношений фрагментировано по другому атрибуту. Производная горизонтальная фрагментация – еще один способ распределить два отношения с целью улучшить совместную обработку выборки и соединения. Обычно если отношение R подвергнуто производной горизонтальной фрагментации, индуцированной отношением S, то фрагменты R и S с одинаковыми значениями атрибута соединения размещаются в одном и том же узле. Кроме того, S можно фрагментировать по предикату выборки.

Поскольку кортежи R размещаются в соответствии с кортежами S, производную фрагментацию следует использовать только для связей один-ко-многим (иерархических) вида S → R, когда кортеж S сопоставляется с *n* кортежами R, но каждому кортежу R соответствует ровно один кортеж S. Заметим, что производную фрагментацию можно было бы использовать для связей многие-ко-многим, при условии что кортежи S (которым соответствует *n* кортежей R) реплицированы. Для простоты мы будем предполагать (и советуем вам поступать именно так), что производная фрагментация применяется только для иерархических связей.

Пример 4.8. Если дана связь один-ко-многим EMP с ASG, то отношение ASG(ENO, PNO, RESP, DUR) можно косвенно фрагментировать по следующим правилам:

$$\mathrm{ASG}_1 = \mathrm{ASG} \ltimes_{\mathrm{ENO}} \mathrm{EMP}_1;$$
$$\mathrm{ASG}_2 = \mathrm{ASG} \ltimes_{\mathrm{ENO}} \mathrm{EMP}_2.$$

Напомним (см. главу 2), что EMP_1 и EMP_2 фрагментированы следующим образом:

$$\mathrm{EMP}_1 = \sigma_{\mathrm{TITLE}=\text{"Programmer"}}(\mathrm{EMP});$$
$$\mathrm{EMP}_2 = \sigma_{\mathrm{TITLE}\neq\text{"Programmer"}}(\mathrm{EMP}).$$

Программа материализации для горизонтально фрагментированного отношения представляет собой объединение фрагментов. В нашем примере имеем

$$\mathrm{ASG} = \mathrm{ASG}_1 \cup \mathrm{ASG}_2.$$ ♦

Запросы к производным фрагментам тоже можно редуцировать. Поскольку этот тип фрагментации полезен для оптимизации запросов с соединением, то было бы хорошей трансформацией заменить соединение с объединением на объединение соединений и применить сформулированное выше правило 2. Поскольку правила фрагментации показывают, какие кортежи

соответствуют друг другу, некоторые соединения будут порождать пустые отношения, если предикаты фрагментации конфликтуют. Например, предикаты ASG_1 и EMP_2 конфликтуют, т. е. мы имеем

$$ASG_1 \bowtie EMP_2 = \emptyset.$$ ♦

В отличие от рассмотренной выше редукции с соединением, здесь редуцированный запрос всегда предпочтительнее фрагментарного, потому что количество частичных соединений обычно равно количеству фрагментов R.

Пример 4.9. Редукцию посредством производной фрагментации мы проиллюстрируем, применив ее к следующему SQL-запросу, который выбирает все атрибуты кортежей из EMP и ASG с одинаковым значением ENO и должностью "Mech. Eng.":

```
SELECT *
FROM   EMP NATURAL JOIN ASG
WHERE  TITLE = "Mech. Eng."
```

Фрагментарный запрос на определенных выше фрагментах EMP_1, EMP_2, ASG_1 и ASG_2 показан на рис. 4.6a. Опустив выборку вниз к фрагментам EMP_1 и EMP_2, мы редуцируем запрос к форме на рис. 4.6b. Действительно, поскольку предикат выборки конфликтует с предикатом EMP_1, фрагмент EMP_1 можно исключить. Чтобы выявить конфликтующие предикаты соединения, заменим соединения с объединениями на объединения соединений. Тогда получится дерево, показанное на рис. 4.6c. Левое поддерево соединяет два фрагмента, ASG_1 и EMP_2, условия которых конфликтуют, потому что ASG_1 определен предикатом TITLE = "Programmer", а EMP_2 – предикатом TITLE ≠ "Programmer". Поэтому левое поддерево, порождающее пустое отношение, можно исключить, и получается редуцированный запрос на рис. 4.6d. Результирующий запрос стал проще, что доказывает ценность фрагментации для повышения производительности распределенных запросов. ♦

4.2.5. Редукция для гибридной фрагментации

Гибридная фрагментация получается комбинированием рассмотренных выше функций фрагментации. Цель гибридной фрагментации – эффективно поддержать запросы, включающие проекцию, выборку и соединение. Заметим, что оптимизация какого-то оператора или комбинации операторов всегда производится за счет других операторов. Например, в случае гибридной фрагментации, основанной на выборке-проекции, одна лишь выборка или одна лишь проекция станет менее эффективной, чем в случае горизонтальной (или вертикальной) фрагментации. В программе материализации для гибридно фрагментированного отношения используются объединения и соединения фрагментов.

Пример 4.10. Ниже приведен пример гибридной фрагментации отношения EMP:

$$EMP_1 = \sigma_{ENO \leq "E4"}(\_ENO,ENAME(EMP));$$

$EMP_2 = \sigma_{ENO>"E4"}(\_ENO,ENAME(EMP))$;
$EMP_3 = \prod_{ENO,TITLE}(EMP)$.

В нашем примере программа материализации имеет вид:

$EMP = (EMP_1 \cup EMP_2) \bowtie_{ENO} EMP_3$. ♦

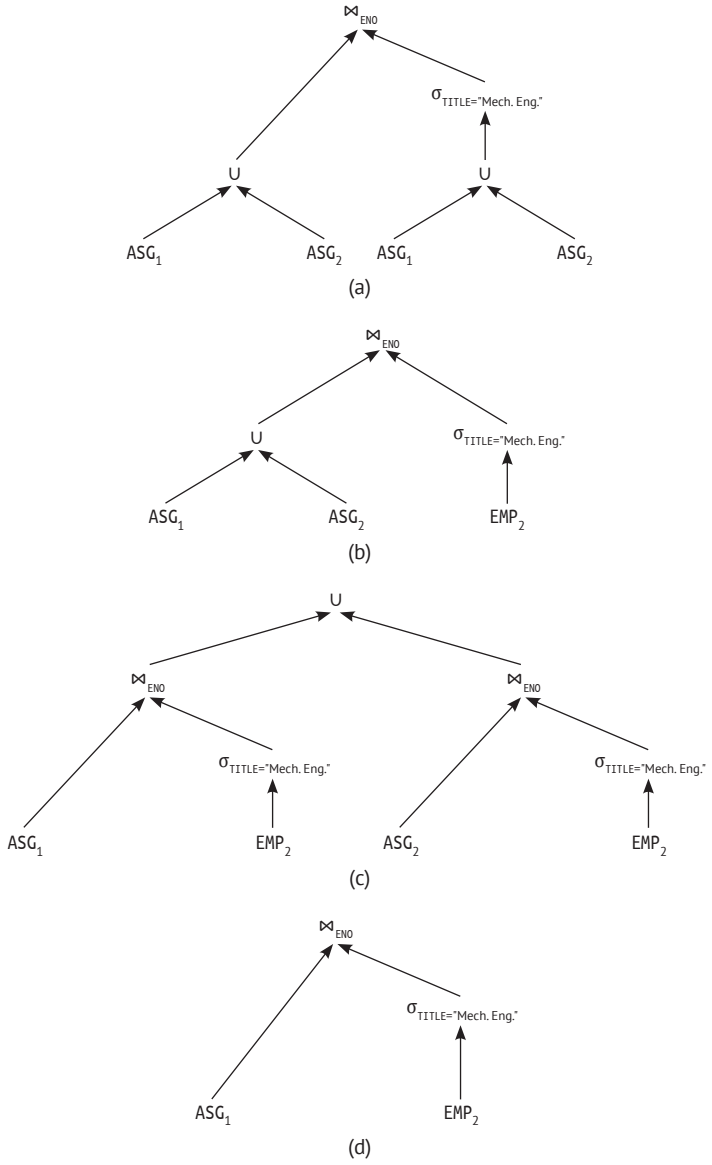


Рис. 4.6 ❖ Редукция для косвенной фрагментации:
(a) фрагментарный запрос; (b) запрос после опускания выборки вниз;
(c) запрос после подъема объединений вверх; (d) редуцированный запрос
после исключения левого поддерева

Запросы к гибридным фрагментам можно редуцировать, комбинируя правила, используемые соответственно для главной горизонтальной, вертикальной и производной горизонтальной фрагментаций. Эти правила можно подытожить следующим образом:

1) исключить пустые отношения, сгенерированные противоречивыми операторами выборки из горизонтальных фрагментов;
2) исключить бесполезные отношения, сгенерированные проекциями на вертикальные фрагменты;
3) заменить соединения с объединениями на объединения соединений, чтобы выделить и исключить бесполезные соединения.

Пример 4.11. Следующий SQL-запрос иллюстрирует применение правил (1) и (2) к приведенной выше горизонтально-вертикальной фрагментации отношения EMP на EMP_1, EMP_2 и EMP_3:

```
SELECT ENAME
FROM   EMP
WHERE  ENO="E5"
```

Фрагментарный запрос на рис. 4.7a можно редуцировать, если сначала протолкнуть выборку вниз, исключив фрагмент EMP, а затем протолкнуть вниз проекцию, исключив фрагмент EMP_3. Редуцированный запрос показан на рис. 4.7b. ♦

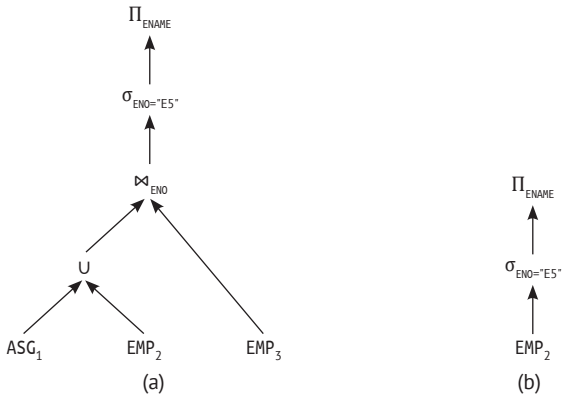


Рис. 4.7 ❖ Редукция для гибридной фрагментации:
(a) фрагментарный запрос; (b) редуцированный запрос

4.3. Порядок соединений в распределенных запросах

Определение порядка соединений – важный аспект централизованной оптимизации запросов. В распределенном контексте порядок соединений еще важнее, потому что соединение фрагментов может увеличить время, затра-

чиваемое на передачу данных. Поэтому пространство поиска, исследуемое оптимизатором распределенных запросов, сконцентрировано на деревьях соединения (см. следующий раздел). Есть два основных подхода к упорядочению соединений в распределенных запросах. В одном производится попытка упорядочить соединения непосредственно, а в другом соединения заменяются комбинациями полусоединений, чтобы минимизировать затраты на передачу данных.

4.3.1. Деревья соединений

Планы выполнения запросов обычно абстрагируются с помощью деревьев операторов, которые определяют порядок выполнения операторов. Они аннотируются дополнительной информацией, например о лучшем алгоритме, выбранном для каждого оператора. Таким образом, для каждого запроса пространство поиска можно определить как множество эквивалентных деревьев операторов, которые можно породить применением правил преобразования. Для характеристики оптимизаторов запросов полезно сосредоточиться на *деревьях соединений*, т. е. деревьях операторов, состоящих только из соединений и декартовых произведений. Дело в том, что изменение порядка соединений – важнейший фактор производительности реляционных запросов.

Пример 4.12. Рассмотрим следующий запрос:

```
SELECT ENAME, RESP
FROM   EMP NATURAL JOIN ASG NATURAL JOIN PROJ
```

На рис. 4.8 показаны три эквивалентных дерева соединений для этого запроса, полученных благодаря ассоциативности бинарных операторов. Каждому дереву можно сопоставить стоимость, основанную на оценке стоимости каждого оператора. Стоимость дерева соединений (c), начинающегося декартовым произведением, вероятно, будет гораздо выше, чем для остальных. ♦

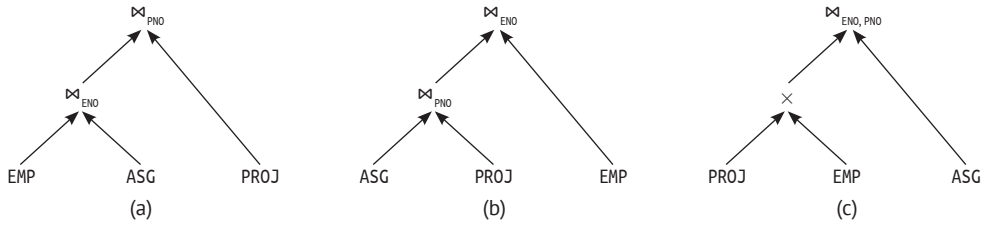


Рис. 4.8 ❖ Три эквивалентных дерева соединений

Для сложного запроса (с большим числом отношений и операторов) количество эквивалентных деревьев операторов может быть очень велико. Например, число альтернативных деревьев соединения, порождаемых применением правил коммутативности и ассоциативности, имеет порядок $O(N!)$ для N отношений. Исследование большого пространства состояний может увеличить время оптимизации, так что оно окажется больше самого времени

выполнения. Поэтому оптимизаторы запросов обычно ограничивают размер рассматриваемого пространства поиска. Первое ограничение связано с использованием эвристик. Самая распространенная эвристика – производить выборку и проекцию при доступе к базовым отношениям. Еще одна полезная эвристика – избегать декартовых произведений, не обязательных для запроса. Например, на рис. 4.8 дерево операторов (c) не войдет в пространство поиска, рассматриваемое оптимизатором.

Еще одно важное ограничение касается формы дерева соединений. Обычно выделяют два вида таких деревьев: линейное и кустистое (рис. 4.9). *Линейным* называется дерево, в котором хотя бы один операнд в каждом операторном узле является базовым отношением. *Леволинейным* деревом называется линейное дерево, в котором правое поддерево узла соединения всегда является листовым узлом, соответствующим базовому отношению. В более общем *кустистом дереве* могут существовать операторы, среди операндов которых нет базовых отношений (т. е. оба операнда – промежуточные отношения). Благодаря рассмотрению только линейных деревьев размер пространства поиска сокращается до $O(2^N)$. Однако в распределенной среде кустистые деревья полезны, поскольку допускают большую степень распараллеливания. Например, в дереве соединений на рис. 4.9(b) операторы $R_1 \bowtie R_2$ и $R_3 \bowtie R_4$ можно вычислять параллельно.

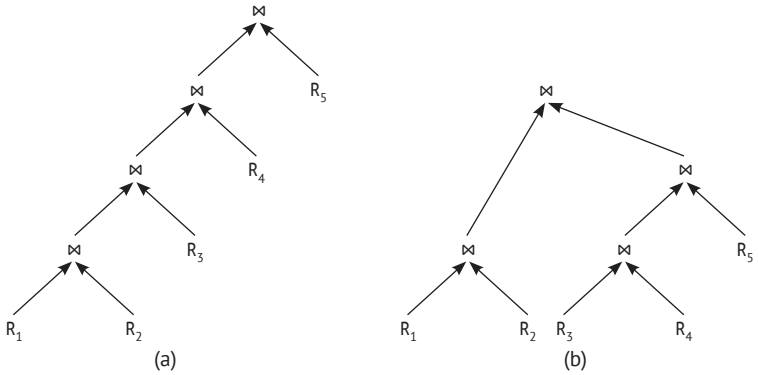


Рис. 4.9 ❖ Две основные формы деревьев соединений:
(a) линейное дерево соединений; (b) кустистое дерево соединений

4.3.2. Порядок соединений

Некоторые алгоритмы оптимизируют порядок соединений напрямую, не используя полусоединения. Цель этого раздела – ясно описать трудности, возникающие в связи с порядком соединения, и подготовить почву для следующего раздела, в котором рассматривается применение полусоединений для оптимизации запросов с соединением.

Чтобы сконцентрироваться на главном, необходимо сделать ряд допущений. Поскольку запрос формулируется для фрагмента, нет нужды различать фрагменты одного и разных отношений. Чтобы упростить нотацию, будем

использовать термин *отношение* для обозначения фрагмента, хранящегося в конкретном узле. Кроме того, чтобы сконцентрироваться на порядке соединений, будем игнорировать время локальной обработки и предполагать, что редукторы (выборка, проекция) исполняются локально либо до, либо во время соединения (напомним, что выполнять выборку в самом начале не всегда эффективно). Поэтому будем рассматривать только запросы с соединениями, в которых отношения-операнды хранятся в разных узлах. Предполагается, что передача отношений производится в режиме «множество за раз», а не «кортеж за раз». Наконец, будем игнорировать время передачи данных в результирующий узел.

Сначала разберем более простую проблему передачи операнда в запросе с одним соединением. Запрос имеет вид $R \bowtie S$, где R и S – отношения, хранящиеся в разных узлах. Очевидно, выгоднее передать меньшее отношение в узел, где находится большее, и, следовательно, возникает две возможности, показанные на рис. 4.10. Чтобы сделать выбор, нам нужно вычислить размеры R и S (предполагается, что это делает функция *size*()). Теперь рассмотрим случай, когда соединяется больше двух отношений. Как и в случае одного соединения, цель алгоритма упорядочения соединений – передавать операнды меньшего размера. Трудность же в том, что операторы соединения могут уменьшать или увеличивать размер промежуточных результатов. Поэтому оценка размера соединения оказывается обязательной, но при этом трудной задачей. Решение состоит в том, чтобы оценить затраты на передачу данных для всех возможных стратегий и выбрать наилучшую. Однако, как мы уже говорили, количество стратегий быстро растет с ростом числа отношений. Этот подход приводит к увеличению стоимости оптимизации, хотя накладные расходы быстро амортизируются, если запрос выполняется часто.

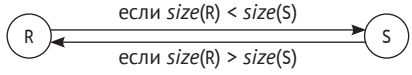


Рис. 4.10 ❖ Передача операндов бинарного оператора

Пример 4.13. Рассмотрим следующий запрос, выраженный на языке реляционной алгебры:

$$PROJ \bowtie_{PNO} ASG \bowtie_{ENO} EMP.$$

Этот запрос можно представить графом соединений, показанным на рис. 4.11. Отметим, что мы сделали некоторые предположения о местоположении всех трех отношений. Запрос можно выполнить по крайней мере пятью разными способами. Мы опишем эти стратегии с помощью следующих программ, где ($R \rightarrow$ узел j) означает «отношение R передается в узел j».

1. $EMP \rightarrow$ узел 2;
   узел 2 вычисляет $EMP' = EMP \bowtie ASG$;
   $EMP' \rightarrow$ узел 3;
   узел 3 вычисляет $EMP' \bowtie PROJ$.

2. ASG → узел 1;
   узел 1 вычисляет EMP′ = EMP ⋈ ASG;
   EMP′ → узел 3;
   узел 3 вычисляет EMP′ ⋈ PROJ.
3. ASG → узел 3;
   узел 3 вычисляет ASG′ = ASG ⋈ PROJ;
   ASG′ → узел 1;
   узел 1 вычисляет ASG′ ⋈ EMP.
4. PROJ → узел 2;
   узел 2 вычисляет PROJ′ = PROJ ⋈ ASG;
   PROJ′ → узел 1;
   узел 1 вычисляет PROJ′ ⋈ EMP.
5. EMP → узел 2;
   PROJ → узел 2;
   узел 2 вычисляет EMP ⋈ PROJ ⋈ ASG;

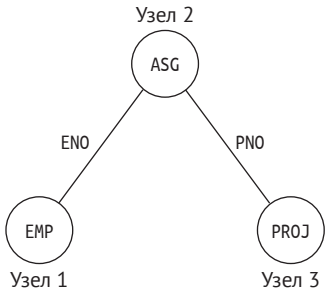


Рис. 4.11 ❖ Граф соединений для распределенного запроса

Чтобы выбрать одну из этих программ, нужно знать или уметь предсказывать следующие размеры: *size*(EMP), *size*(ASG), *size*(PROJ), *size*(EMP ⋈ ASG) и *size*(ASG ⋈ PROJ). Кроме того, если интерес представляет прежде всего время ответа, то при оптимизации следует учитывать, что для стратегии 5 можно выполнить две операции передачи данных параллельно. Альтернатива перечислению всех решений – воспользоваться эвристикой, которая учитывает только размеры отношений-операндов, предположив, к примеру, что мощность результирующего соединения равна произведению мощностей операндов. В этом случае отношения упорядочиваются по возрастанию размеров, а порядок выполнения определяется этим упорядочением и графом соединений. Например, для порядка (EMP, ASG, PROJ) можно было бы использовать стратегию 1, а для порядка (PROJ, ASG, EMP) – стратегию 4. ♦

4.3.3. Алгоритмы на основе полусоединений

У оператора полусоединения есть важное свойство – он уменьшает размер отношения-операнда. Если процессор запросов считает, что самая дорогая составляющая стоимости – коммуникация, то полусоединение особенно по-

лезно для улучшения обработки операторов распределенного соединения, поскольку уменьшает объем данных, передаваемых между узлами. Однако при использовании полусоединений может увеличиться количество сообщений и время локальной обработки. В ранних распределенных СУБД, например SDD-1, которые проектировались для медленных глобальных сетей, полусоединения применялись очень широко. Впрочем, они сохраняют привлекательность и в контексте быстрых сетей, когда приводят к сильной редукции операнда соединения. Поэтому некоторые алгоритмы стремятся найти оптимальное сочетание соединений и полусоединений.

В этом разделе мы покажем, как оператор полусоединения можно использовать для уменьшения полного времени обработки запроса с соединением. Делаются те же предположения, что в разделе 4.3.2. Основной недостаток подхода к соединению, описанного в предыдущем разделе, – тот факт, что отношения-операнды нужно передавать между узлами целиком. Полусоединение играет роль редуктора размера отношения – как выборка.

Соединение по атрибуту A двух отношений R и S, хранящихся в узлах 1 и 2 соответственно, можно вычислить заменой одного или обоих отношений-операндов полусоединением с другим отношением, применив следующие правила:

$$\begin{aligned} R \bowtie_A S &\Leftrightarrow (R \ltimes_A S) \bowtie_A S \\ &\Leftrightarrow R \bowtie_A (S \ltimes_A R) \\ &\Leftrightarrow (R \ltimes_A S) \bowtie_A (S \ltimes_A R). \end{aligned}$$

Для выбора одной из трех стратегий полусоединения нужно оценить сопутствующие затраты.

Использовать полусоединение выгодно, если стоимость его порождения и передачи на другой узел меньше стоимости передачи всего отношения-операнда и выполнения фактического соединения. Для иллюстрации потенциального выигрыша от использования полусоединения сравним стоимости двух альтернатив, $R \bowtie_A S$ и $(R \ltimes_A S) \bowtie_A S$, в предположении, что *size*(R) < *size*(S).

В следующей программе, написанной в обозначениях раздела 4.3.2, используется оператор полусоединения:

1. $\prod_A(S) \rightarrow$ узел 1.
2. Узел 1 вычисляет $R' = R \ltimes_A S$.
3. $R' \rightarrow$ узел 2.
4. Узел 2 вычисляет $R \ltimes_A S$.

Для простоты не будем обращать внимания на константу T_{MSG} во времени передачи данных, предположив, что член $T_{TR} * size(R)$ намного больше. Тогда можно сравнить оба варианта с точки зрения размера передаваемых данных. Стоимость алгоритма, основанного на соединении, равна стоимости передачи отношения R на узел 2. Стоимость алгоритма, основанного на полусоединении, равна стоимости шагов 1 и 3 выше. Поэтому подход на основе полусоединения лучше, если

$$size(\prod_A(S)) + size(R \ltimes_A S) < size(R).$$

Подход на основе полусоединения лучше, если полусоединение работает как редуктор, т. е. если в соединении участвует немного кортежей R. Подход

на основе соединения лучше, если в соединении участвуют почти все кортежи R, поскольку для полусоединения необходима дополнительная передача проекции на атрибут соединения. Стоимость шага проекции можно снизить, закодировав результат проекции в виде битовых массивов и тем самым уменьшив стоимость передачи значений соединенных атрибутов. Важно отметить, что ни тот, ни другой подход не является безусловно предпочтительным, их следует рассматривать как дополняющие друг друга.

Вообще, полусоединение может принести пользу в плане уменьшения размера отношений-операндов, участвующих в запросах с несколькими соединениями. Однако в таких случаях оптимизация запроса становится более сложной. Снова рассмотрим граф соединений отношений EMP, ASG и PROJ на рис. 4.11. Мы можем применить описанный выше алгоритм соединения, применив полусоединения к каждому отдельному соединению. Таким образом, программа вычисления $\text{EMP} \bowtie \text{ASG} \bowtie \text{PROJ}$ имеет вид $\text{EMP}' \bowtie \text{ASG}' \bowtie \text{PROJ}$, где $\text{EMP}' = \text{EMP}' \ltimes \text{ASG}$ и $\text{ASG}' = \text{ASG} \ltimes \text{PROJ}$.

Однако можно еще уменьшить размер отношения-операнда, воспользовавшись не одним, а несколькими полусоединениями. Например, EMP' в предыдущей программе можно заменить на EMP'', вычисляемый по формуле

$$\text{EMP}'' = \text{EMP} \ltimes (\text{ASG} \ltimes \text{PROJ}),$$

т. к. если $size(\text{ASG} \ltimes \text{PROJ}) \leqslant size(\text{ASG})$, то $size(\text{EMP}'') \leqslant size(\text{EMP}')$. Таким образом, EMP можно редуцировать с помощью последовательности полусоединений: $\text{EMP} \ltimes (\text{ASG} \ltimes \text{PROJ})$. Такая последовательность полусоединений называется *программой полусоединений* для EMP. Подобные программы полусоединений можно найти для любого отношения, участвующего в запросе. Например, PROJ можно редуцировать с помощью программы $\text{PROJ} \ltimes (\text{ASG} \ltimes \text{EMP})$. Но не все участвующие в запросе отношения нужно редуцировать; в частности, можно игнорировать отношения, не участвующие в окончательных соединениях.

Для данного отношения существует несколько потенциальных программ полусоединений. На самом деле количество вариантов экспоненциально зависит от количества отношений. Но существует одна оптимальная программа полусоединений, называемая *полным редуктором*, которая редуцирует каждое отношение R больше, чем другие. Проблема в том, чтобы найти полный редуктор. Простой способ заключается в вычислении уменьшения размера для всех возможных программ полусоединений и выборе наилучшей. Но этот метод перебора сталкивается с двумя трудностями:

1) существует класс так называемых *циклических запросов*, для которых граф соединений содержит циклы и найти полный редуктор невозможно;
2) для остальных запросов, называемых *древовидными*, полные редукторы существуют, но количество потенциальных программ полусоединений экспоненциально зависит от числа отношений, что делает задачу перебора NP-трудной.

Далее мы обсудим решения этих проблем.

Пример 4.14. Предположим, что мы добавили атрибут CITY в отношения EMP (переименовано в ET), PROJ (переименовано в PT) и ASG (переименовано в AT).

Атрибут CITY в отношении AT соответствует городу, в котором проживает работник с идентификатором ENO.

```
ET(ENO, ENAME, TITLE, CITY)
AT(ENO, PNO, RESP, DUR, CITY)
PT(PNO, PNAME, BUDGET, CITY)
```

Следующий SQL-запрос выбирает названия проектов и имена всех работников, проживающих в том же городе, в котором выполняется их проект.

```
SELECT ENAME, PNAME
FROM   ET NATURAL JOIN AT NATURAL JOIN PT
       NATURAL JOIN ET
```

Как показано на рис. 4.12a, этот запрос циклический. ♦

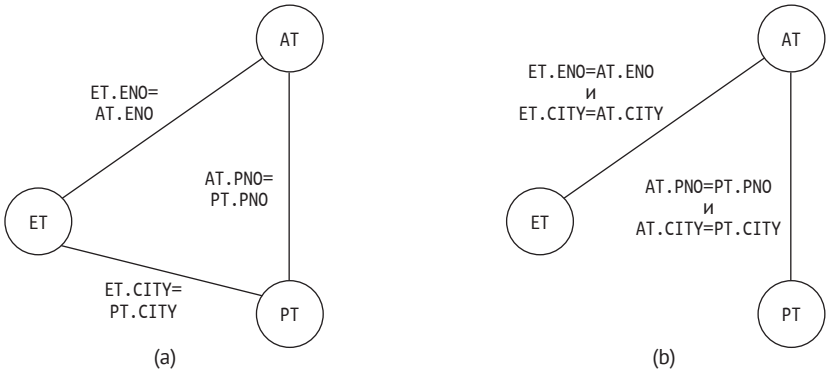


Рис. 4.12 ❖ Преобразование циклического запроса:
(a) циклический запрос; (b) эквивалентный ациклический запрос

Для запроса из примера 4.14 полного редуктора не существует. На самом деле можно построить программы полусоединений для его редуцирования, но число операторов умножается на число кортежей в каждом отношении, из-за чего этот подход становится неэффективным. Возможное решение – преобразовать циклический граф в дерево, удалив из графа одно ребро и добавив подходящие предикаты к другим ребрам, так чтобы удаленный предикат сохранился в силу транзитивности. В примере на рис. 4.12b, где удалено ребро (ET, PT), дополнительным предикатом является ET.CITY = AT.CITY, а AT.CITY = PT.CITY влечет за собой ET.CITY = PT.CITY в силу транзитивности. Стало быть, ациклический запрос эквивалентен циклическому.

Хотя полные редукторы для древовидных запросов существуют, задача их нахождения является NP-трудной. Однако существует важный класс *цепных запросов*, для которых имеется полиномиальный алгоритм. Отношения в графе соединений цепного запроса можно упорядочить, так что каждое отношение соединяется только со следующим отношением в порядке следования. Кроме того, результат запроса находится в конце цепочки. Например, запрос на рис. 4.11 цепной. Из-за трудности реализации алгоритма с полны-

ми редукторами в большинстве систем для уменьшения размера отношения используются одиночные полусоединения.

4.3.4. Сравнение соединения и полусоединения

По сравнению с соединением полусоединение вовлекает больше операторов, но, возможно, с меньшими операндами. На рис. 4.13 показаны эти различия на примере эквивалентных стратегий соединения и полусоединения для запроса с графом соединений, изображенным на рис. 4.11. Для соединения EMP ⋈ ASG нужно передать одно отношение, например ASG, в узел другого отношения EMP и завершить соединение локально. Но при использовании полусоединения передачи отношения ASG удается избежать. Вместо этого передаются значения атрибутов соединения отношения EMP в узел отношения ASG, а затем производится передача соответственных кортежей отношения ASG в узел отношения EMP, где соединение и завершается. Если у полусоединения высокая избирательность, то такой подход может значительно сэкономить на времени передачи данных. Кроме того, подход на основе полусоединения может уменьшить локальное время обработки за счет использования индексов по атрибуту соединения. Снова рассмотрим соединение EMP ⋈ ASG в предположении, что имеется выборка из ASG и по атрибуту соединения построен индекс над ASG. Без полусоединения мы должны были бы сначала выполнить выборку из ASG, а затем отправить результирующее отношение в узел EMP, где соединение завершается. Таким образом, индекс по атрибуту соединения над ASG не используется (потому что соединение производится в узле EMP). Если же выбрать подход на основе полусоединения, то и выборка, и полусоединение ASG ⋉ EMP производятся в узле ASG и могут быть выполнены эффективно с применением индексов.

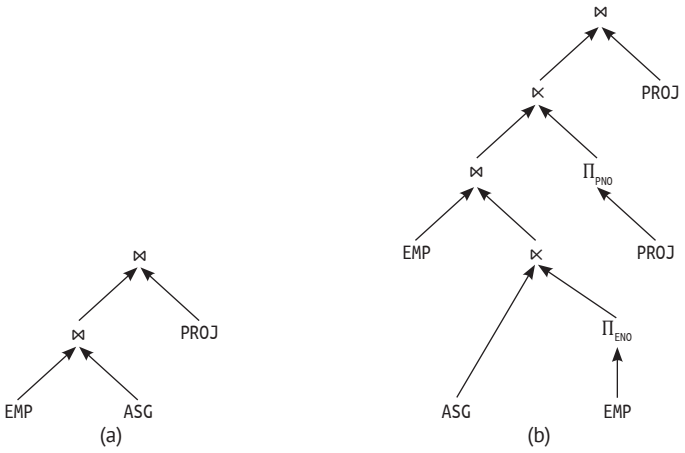


Рис. 4.13 ❖ Сравнение соединения и полусоединения:
(a) соединение; (b) полусоединение

Полусоединения могут принести выигрыш даже при быстрых сетях, если обладают очень высокой избирательностью и реализованы с помощью битовых массивов. Битовый массив $BA[1 : n]$ полезен для кодирования значений атрибута соединения в одном отношении. Рассмотрим полусоединение $R \ltimes S$. Тогда бит $BA[i]$ равен 1, если существует значение атрибута соединения $A = val$ в отношении S такое, что $h(val) = i$, где h – некоторая хеш-функция. В противном случае $BA[i]$ равен 0. Такой битовый массив гораздо меньше, чем список значений атрибута соединения. Поэтому передача битового массива вместо значений атрибута соединения в узел отношения R экономит время передачи. Завершить полусоединение можно следующим образом. Каждый кортеж отношения R, в котором значение атрибута соединения равно *val*, принадлежит полусоединению, если $BA[h(val)] = 1$.

4.4. Распределенная модель стоимости

Модель стоимости оптимизатора включает функции для предсказания стоимости операторов, статистику и базовые данные, а также формулы для вычисления размеров промежуточных результатов. Стоимость выражается в терминах времени выполнения, поэтому функция стоимости представляет время выполнения запроса.

4.4.1. Функции стоимости

Стоимость стратегии распределенного выполнения можно выразить как полное время или как время ответа. Полное время равно сумме всех временных (они же стоимостные) составляющих, а время ответа – это время, прошедшее с момента начала выполнения до момента завершения запроса. Общая формула для вычисления полного времени имеет вид:

$$Total\_time = T_{CPU} * \#insts + T_{I/O} * \#I/Os + T_{MSG} * \#msgs + T_{TR} * \#bytes.$$

Первые две составляющие измеряют время локальной обработки, где T_{CPU} – время выполнения одной команды процессора, а $T_{I/O}$ – время дискового ввода-вывода. Время передачи данных отражают две последние составляющие: T_{MSG} – фиксированное время инициализации и приема сообщения, а T_{TR} – время передачи единицы данных с одного узла на другой. В качестве единицы данных выбран байт (*#bytes* – суммарный размер всех сообщений), но можно взять и другую единицу (например, пакет). Обычно предполагается, что T_{TR} постоянно. Для глобальных сетей, в которых расстояние между узлами сильно различается, это может быть и не так. Однако такое предположение значительно упрощает оптимизацию запроса. В этом случае считается, что время передачи *#bytes* байтов с одного узла на другой – линейная функция *#bytes*:

$$CT(\#bytes) = T_{MSG} + T_{TR} * \#bytes.$$

Стоимость в общем случае выражается в терминах единиц времени и может быть переведена в другие единицы (например, доллары).

Относительные значения стоимостных коэффициентов характеризуют распределенную среду, в которой работает база данных. Топология сети сильно влияет на соотношение составляющих. В глобальной сети, например в интернете, обычно преобладает время передачи данных, в локальных сетях составляющие сбалансированы лучше. Поэтому в большинстве ранних распределенных СУБД, проектировавшихся для глобальных сетей, время локальной обработки игнорировалось, а все силы направлялись на минимизацию затрат на коммуникацию. С другой стороны, при проектировании распределенных СУБД для локальных сетей учитываются все три составляющие. Новые более быстрые сети (глобальные и локальные) позволили сократить относительные затраты на коммуникацию при прочих равных условиях. Тем не менее коммуникация остается доминирующим фактором в глобальных сетях, поскольку данные приходится передавать на большие расстояния.

Если целевой функцией оптимизатора является время ответа на запрос, то следует учитывать параллельную локальную обработку и параллельную передачу данных. Общая формула вычисления времени ответа имеет вид

$$Response\_time = T_{CPU} * seq\_\#insts + T_{I/O} * seq\_\#I/Os + T_{MSG} * seq\_\#msgs + T_{TR} * seq\_\#bytes,$$

где *seq_#x* обозначает максимальное значение *x* (количество команд, если *x* равно *insts*, количество операций ввода-вывода, если *x* равно *I/O*, количество сообщений, если *x* равно *msgs*, и количество байтов, если *x* равно *bytes*) в одной последовательной выполняемой части запроса. Таким образом, вся параллельная обработка и передача данных игнорируются.

Пример 4.15. Проиллюстрируем различие между полной стоимостью и временем ответа на примере рис. 4.14, где в узле 3 вычисляется ответ на запрос к данным в узлах 1 и 2. Для простоты предположим, что в расчет принимается только стоимость коммуникации.

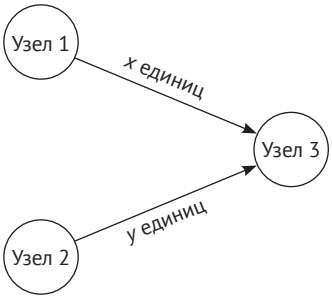


Рис. 4.14 ❖ Пример передачи данных для выполнения запроса

Предположим, что T_{MSG} и T_{TR} выражены в единицах времени. Полное время передачи *x* единиц данных из узла 1 на узел 3 и *y* единиц данных из узла 2 на узел 3 равно

$Total\_time = 2\ T_{MSG} + T_{TR} * (x + y)$.

Время ответа на тот же самый запрос можно приближенно записать в виде

$Response\_time = max\{T_{MSG} + T_{TR} * x, T_{MSG} + T_{TR} * y\}$,

поскольку операции передачи выполняются параллельно. ♦

Минимизация времени ответа достигается путем увеличения степени параллелизма. Но это не означает, что и полное время также минимизируется. Напротив, полное время может увеличиться, например из-за большего объема локальной обработки и передаваемых данных. Для минимизации полного времени требуется улучшить использование ресурсов и тем самым увеличить пропускную способность системы. На практике желательно найти компромисс между тем и другим. В разделе 4.5 мы приведем алгоритмы, которые могут оптимизировать сочетание полного времени и времени ответа с упором на то или другое.

4.4.2. Статистика базы данных

Основной фактор, влияющий на производительность стратегии выполнения, – размер порождаемых при этом промежуточных отношений. Если следующий оператор находится в другом узле, то промежуточное отношение нужно передавать по сети. Поэтому так важно оценить размер промежуточных результатов операторов реляционной алгебры с целью минимизировать объем передаваемых данных. Эта оценка основана на статистической информации о базовых отношениях и формулах для предсказания мощности результатов реляционных операторов. Существует прямая связь между точностью статистики и затратами на управление ей – чем статистика точнее, тем дороже она обходится. Для отношения R, разбитого на фрагменты R_1, R_2, …, R_r, обычно собирается следующая статистика:

1) для каждого атрибута A отношения R: его длина (в байтах), обозначаемая *length*(A), мощность области определения (домена), обозначаемая *card*(*dom*[A]), которая равна числу уникальных значений в *dom*[A], а в случае, когда домен является упорядоченным множеством (например, множеством целых или вещественных чисел), – также минимально и максимально возможные значения, обозначаемые min(A) и max(A);
2) для каждого атрибута A каждого фрагмента R_i: количество различных значений A, т. е. мощность проекции фрагмента R_i на A, обозначаемая $card(\prod_A(R_i))$;
3) количество кортежей в каждом фрагменте R_i, обозначаемое $card(R_i)$.

Кроме того, для каждого атрибута A может храниться гистограмма, аппроксимирующая распределение частот атрибута количеством интервалов, каждый из которых соответствует диапазону значений.

Эта статистика полезна для предсказания размера промежуточных отношений. Напомним, что в главе 2 мы определили размер промежуточного отношения R следующим образом:

$$size(\mathtt{R}) = card(\mathtt{R}) * length(\mathtt{R}),$$

где *length*(R) – длина кортежа R в байтах, а *card*(R) – количество кортежей в R.

Для оценки *card*(R) нужны формулы. Обычно о базе данных делается два упрощающих предположения. Предполагается, что распределение значений атрибутов отношения равномерно и что все атрибуты независимы, т. е. значение одного атрибута не влияет на значение любого другого. На практике оба предположения часто неверны, но благодаря им задача становится решаемой. Исходя из этих предположений, мы можем вывести простые формулы для оценивания мощности результатов основных операторов реляционной алгебры, зная их избирательность. *Коэффициент избирательности* оператора, т. е. доля кортежей отношения-операнда, участвующих в результате операции, обозначается *SF*(*op*), где *op* – операция. Это вещественное значение в диапазоне от 0 до 1. Если значение мало (например, 0.001), то избирательность хорошая (высокая), а если велико (например, 0.5), то избирательность плохая (низкая).

Рассмотрим два самых распространенных оператора – выборку и соединение. Мощность выборки равна

$$card(\sigma_{\mathtt{F}}(\mathtt{R})) = SF(\sigma_{\mathtt{F}}(\mathtt{R})) * card(\mathtt{R}),$$

где $SF(\sigma_{\mathtt{F}}(\mathtt{R}))$ следующим образом вычисляется для базовых предикатов:

$$SF(\sigma_{\mathtt{A}=value}(\mathtt{R})) = \frac{1}{card(\Pi_{\mathtt{A}}(\mathtt{R}))};$$

$$SF(\sigma_{\mathtt{A}>value}(\mathtt{R})) = \frac{\max(\mathtt{A}) - value}{\max(\mathtt{A}) - \min(\mathtt{A})};$$

$$SF(\sigma_{\mathtt{A}<value}(\mathtt{R})) = \frac{value - \min(\mathtt{A})}{\max(\mathtt{A}) - \min(\mathtt{A})}.$$

Мощность соединения равна

$$card(\mathtt{R} \bowtie \mathtt{S}) = SF(\mathtt{R} \bowtie_{\mathtt{A}} \mathtt{S}) * card(\mathtt{R}) * card(\mathtt{S}).$$

Не существует общего способа оценить $SF(\mathtt{R} \bowtie_{\mathtt{A}} \mathtt{S})$, не имея дополнительной информации. Простая аппроксимация – считать этот коэффициент постоянным, например 0.01, что отражает априорное знание об избирательности соединения. Однако есть случай, и довольно часто встречающийся, когда оценка точна. Если отношение R соединено с S по равенству атрибута A, являющегося первичным ключом R и внешним ключом S, то коэффициент избирательности соединения можно аппроксимировать выражением

$$SF(\mathtt{R} \bowtie_{\mathtt{A}} \mathtt{S}) = \frac{1}{card(\mathtt{R})},$$

потому что каждый кортеж S сопоставляется самое большее с одним кортежем R.

4.5. Оптимизация распределенных запросов

В этом разделе мы проиллюстрируем технику, представленную в предыдущих разделах, в контексте трех основных алгоритмов оптимизации запросов. Сначала покажем динамический и статический подходы, а затем гибридный.

4.5.1. Динамический подход

Динамический подход мы проиллюстрируем на примере алгоритма из системы Distributed INGRES. Целевая функция алгоритма – минимизация комбинации времени передачи данных и времени ответа. Однако эти две цели могут конфликтовать. Например, при увеличении времени передачи данных (вследствие параллелизма) время ответа вполне может уменьшиться. Таким образом, функция может назначать больший вес одному или другому фактору. Заметим, что алгоритм оптимизации запроса игнорирует затраты на передачу данных узлу, где нужен результат. Алгоритм принимает во внимание фрагментацию, но для простоты рассматривается только горизонтальная фрагментация.

Поскольку рассматриваются как общие, так и широковещательные сети, оптимизатор учитывает топологию сети. В широковещательных сетях одна единица данных может передаваться из одного узла на все остальные за одну операцию, и алгоритм пользуется этой возможностью. Например, широковещание используется для репликации фрагментов и, как следствие, увеличения степени параллелизма.

На вход алгоритма подается запрос, выраженный в терминах реляционного исчисления (в конъюнктивной нормальной форме), и информация о схеме (тип сети, а также местоположение и размер каждого фрагмента). Алгоритм исполняется *главным узлом*, инициировавшим запрос. Алгоритм 4.1 содержит псевдокод алгоритма Dynamic-QOA.

Все запросы с одним отношением (например, выборки и проекции), которые можно отделить, сначала обрабатываются локально (шаг 1). Затем к исходному запросу применяется алгоритм приведения (шаг 2). Приведение – это метод, изолирующий все неприводимые подзапросы и подзапросы с одним отношением путем отделения. Запросы с одним отношением игнорируются, потому что они уже обработаны на шаге 1. Таким образом, процедура REDUCE порождает последовательность неприводимых подзапросов $q_1 \rightarrow q_2 \rightarrow \ldots \rightarrow q_n$ такую, что два соседних подзапроса имеют не более одного общего отношения.

На основе списка неприводимых запросов, изолированных на шаге 2, и размеров фрагментов на шаге 3.1 выбирается следующий подзапрос *MRQ′*, имеющий не менее двух переменных, и к нему применяются шаги 3.2–3.4. Шаги 3.1 и 3.2 обсуждаются ниже. На шаге 3.2 выбирается наилучшая стратегия обработки запроса *MRQ′*. Эта стратегия описывается списком пар (*F*, *S*), в которых *F* – фрагмент, подлежащий передаче на обрабатывающий узел *S*. На шаге 3.3 все фрагменты передаются на обрабатывающие их узлы. Нако-

нец, на шаге 3.4 выполняется запрос *MRQ′*. Если еще остались подзапросы, то алгоритм переходит на шаг 3 и начинает следующую итерацию. В противном случае алгоритм завершается.

Алгоритм 4.1. Dynamic-QOA

Вход: *MRQ*: запрос с несколькими отношениями
Выход: результат последнего запроса с несколькими отношениями
begin
  for *каждого отделимого* ORQ_i *в MRQ* **do** {*ORQ* – запрос с одним отношением}
    run(ORQ_i) (1)
  end for
  {MRQ заменен *n* неприводимыми запросами}
  MRQ′_list ← REDUCE(*MRQ*) (2)
  while $n \neq 0$ **do** {*n* – число неприводимых запросов } (3)
    {выбрать следующий неприводимый запрос, включающий наименьшие фрагменты}
    MRQ′ ← SELECT_QUERY(*MRQ′_list*) (3.1)
    {определить фрагменты для передачи и обрабатывающий узел для *MRQ′*}
    Fragment-site-list ← SELECT_STRATEGY(*MRQ′*) (3.2)
    {переместить выбранные фрагменты на выбранные узлы}
    for *каждой пары* (*F*, *S*) *в списке Fragment-site-list* **do**
      переместить фрагмент *F* на узел S (3.3)
    end for
    выполнить *MRQ′* (3.4)
    $n \leftarrow n - 1$
  end while
  {выходом является результат последнего *MRQ′*}
end

Оптимизация производится на шагах 3.1 и 3.2. Алгоритм породил подзапросы с несколькими компонентами и определил их зависимости (как в реляционно-алгебраическом дереве). На шаге 3.1 в качестве следующего подзапроса просто берется тот, что не имеет предшественников и включает меньшие фрагменты. Это минимизирует размер промежуточных результатов. Например, если у запроса *q* есть три подзапроса q_1, q_2, q_3 с зависимостями $q_1 \rightarrow q_3$, $q_2 \rightarrow q_3$ и если фрагменты, на которые ссылается q_1, меньше тех, на которые ссылается q_2, то выбирается q_1. В зависимости от сети на этот выбор может также влиять количество узлов, где хранятся релевантные фрагменты.

Далее необходимо выполнить выбранный подзапрос. Поскольку отношение, участвующее в подзапросе, может храниться в разных узлах и даже быть фрагментировано, подзапрос, возможно, придется разбить на еще меньшие части.

Пример 4.16. Рассмотрим следующий запрос:

Получить имена работников, трудящихся над проектом CAD/CAM

Этот запрос можно выразить на SQL с помощью следующего запроса q_1 к базе данных конструкторской фирмы:

```
q1 : SELECT EMP.ENAME
     FROM   EMP NATURAL JOIN ASG NATURAL JOIN PROJ
     WHERE  PNAME="CAD/CAM"
```

Предположим, что отношения EMP, ASG и PROJ хранятся, как показано ниже, причем отношение EMP фрагментировано.

Узел 1	Узел 2
EMP_1	EMP_2
ASG	PROJ

Возможно несколько стратегий, в том числе:

1) выполнить весь запрос (EMP ⋈ ASG ⋈ PROJ), переместив EMP_1 и ASG на узел 2;
2) выполнить запрос (EMP ⋈ ASG) ⋈ PROJ, переместив (EMP_1 ⋈ ASG) и ASG на узел 2, и т. д.

Для выбора стратегии необходимо оценить размер промежуточных результатов. Например, если *size*(EMP ⋈ ASG) > *size*(EMP_1), то стратегия 1 предпочтительнее стратегии 2. Так что без оценки размера соединений не обойтись. ♦

На шаге 3.2 решается следующая проблема оптимизации: определить, как выполнять подзапрос, для чего нужно выбрать подлежащие перемещению фрагменты и узлы, в которых будет осуществляться обработка. Для подзапроса с n-отношениями фрагменты $n - 1$ отношений предстоит переместить в узел (или узлы), где находится оставшееся отношение, скажем R, и там реплицировать. Кроме того, оставшееся отношение, возможно, придется разбить на k «уравненных» фрагментов, чтобы повысить степень параллелизма. Этот метод, называемый *фрагментация с репликацией*, выполняет подстановку фрагментов, а не кортежей. Выбор оставшегося отношения и количества обрабатывающих узлов k, на которые его следует фрагментировать, основан на целевой функции и топологии сети. Напомним, репликация дешевле в широковещательных сетях, чем в сетях с двухточечным соединением. Кроме того, выбор количества обрабатывающих узлов включает компромисс между временем ответа и полным временем. При увеличении количества узлов уменьшается время ответа (из-за параллельной обработки), но возрастает полное время, в частности из-за увеличения затрат на передачу данных.

В формулах для минимизации времени передачи данных и времени обработки используются местоположение фрагментов, их размеры и тип сети. Можно минимизировать обе величины, но с предпочтением какой-то одной. Для иллюстрации приведем правила минимизации времени передачи данных. Правило минимизации времени ответа еще сложнее. Введем следующие предположения. В запросе участвует n отношений R_1, R_2, …, R_n. R_i^j обозначает фрагмент R_i, хранящийся в узле j. Всего в сети m узлов. Наконец, CT_k(*#bytes*) обозначает время передачи *#bytes* на k узлов, где $1 \leqslant k \leqslant m$. В правиле минимизации времени передачи данных типы узлов сети рассматриваются по отдельности. Сначала остановимся на широковещательной сети. В этом случае имеем

$CT_k(\#bytes) = CT_1(\#bytes)$.

Правило можно сформулировать в виде

if $\max_{j=1,m}(\sum_{i=1}^{n} size(R_i^j)) > \max_{i=1,n}(size(R_i))$
then
обрабатывающим будет узел j с наибольшим объемом данных
else
R_p наибольшее отношение **and** узел R_p является обрабатывающим

Если неравенство в условии выполнено, то в одном узле данных, полезных для запроса, больше, чем размер самого большого отношения. Тогда этот узел должен быть обрабатывающим. Если же неравенство не выполнено, то размер одного отношения больше, чем максимальный объем полезных данных в одном узле. Поэтому такое отношение должно быть выбрано в качестве R_p, а обрабатывающими узлами будут те, которые содержат его фрагменты.

Теперь рассмотрим сети с двухточечным соединением. Тогда

$CT_k(\#bytes) = k * CT_1(\#bytes)$.

Очевидно, что в качестве R_p, минимизирующего время передачи данных, нужно взять наибольшее отношение. В предположении, что узлы упорядочены по убыванию полезных для запроса данных, т. е.

$$\sum_{i=1}^{n} size(R_i^j) > \sum_{i=1}^{n} size(R_i^{j+1}),$$

количество обрабатывающих узлов k определяется так:

if $\sum_{i \neq p}(size(R_i) - size(R_i^1)) > size(R_p^1)$
then
$k = 1$
else
k – наибольшее j такое, что $\sum_{i \neq p}(size(R_i) - size(R_i^1)) \leqslant size(R_p^j)$

Это правило выбирает узел в качестве обрабатывающего, только если объем данных, которые он должен получить, меньше дополнительного объема данных, которые он должен был бы отправить, если бы не был обрабатывающим. Очевидно, что в части then этого правила предполагается, что в узле 1 хранится фрагмент R_p.

Пример 4.17. Рассмотрим запрос PROJ ⋈ ASG, где PROJ и ASG фрагментированы. Предположим, что размещение и размеры (в килобайтах) фрагментов описываются следующей таблицей:

	Узел 1	Узел 2	Узел 3	Узел 4
PROJ	1000	1000	1000	1000
ASG			2000	

В сети с двухточечным соединением наилучшая стратегия – передать каждый фрагмент $PROJ_i$ на узел 3; при этом понадобится передать 3000 КБ, а не

6000 КБ, как в случае, если бы ASG передавалось на узлы 1, 2 и 4. Однако в широковещательной сети лучшей стратегией было бы передать ASG (за один присест) на узлы 1, 2 и 4, что потребовало бы передачи 2000 КБ. Эта стратегия быстрее и максимизирует время ответа, потому что соединения можно выполнить параллельно. ♦

Этот динамический алгоритм оптимизации запроса характеризуется ограниченным поиском в пространстве решений – решение об оптимизации принимается на каждом шаге без оглядки на его последствия для глобальной оптимизации. Однако алгоритм способен исправить локальное решение, оказавшееся неправильным.

4.5.2. Статический подход

Этот статический подход мы проиллюстрируем на примере алгоритма R*, лежащего в основе многих оптимизаторов распределенных запросов. Он производит исчерпывающий поиск среди всех возможных стратегий, стремясь выбрать ту, у которой наименьшая стоимость. Хотя предсказание и перебор стратегий может стоить дорого, эти накладные расходы быстро амортизируются, если запрос выполняется часто. Компиляция запроса – распределенная задача, которую координирует *главный узел*, инициировавший запрос. Оптимизатор в главном узле принимает все относящиеся к нескольким узлам решения, например выбор фрагментов и исполняющих узлов, а также метод передачи данных. *Подчиненные узлы*, т. е. все прочие узлы, в которых хранятся участвующие в запросе отношения, принимают остальные локальные решения (например, о порядке соединений в узле) и генерируют планы локального доступа для запроса. Целевой функцией оптимизатора является полное время, включая затраты на локальную обработку и передачу данных.

Теперь опишем этот алгоритм оптимизации словами. Входными данными для него являются: фрагментарный запрос, выраженный в виде реляционно-алгебраического дерева запроса, местоположение отношений и их статистика.

Оптимизатор должен выбрать порядок соединений, алгоритм соединений (вложенные циклы или соединение слиянием) и путь доступа к каждому фрагменту (например, по кластерному индексу, последовательный просмотр и т. д.). Эти решения основаны на статистике, формулах для оценки размера промежуточных результатов и информации о пути доступа. Кроме того, оптимизатор должен выбрать узлы результатов соединений и метод передачи данных между узлами. Для соединения двух отношений есть три возможности: узел первого отношения, узел второго отношения или третий узел (т. е. узел третьего отношения, с которым нужно будет соединить результат). Для поддержки межузловых передач есть два метода.

1. *Передать целиком*. Все отношение передается в узел соединения и сохраняется во временном отношении перед выполнением соединения. Если соединение производится слиянием, то сохранять отношение не нужно, и узел соединения может обрабатывать приходящие кортежи в конвейерном режиме, по мере поступления.

2. *Выбирать по мере необходимости*. Внешнее отношение просматривается целиком, и для каждого кортежа значение, по которому производится соединение, передается в узел внутреннего отношения, который выбирает внутренние кортежи, соответствующие этому значению, и передает выбранные кортежи в узел внешнего отношения. Этот метод, называемый также *соединением связыванием* (bind join), эквивалентен полусоединению внутреннего отношения с каждым внешним кортежем.

Алгоритм 4.2. Static*-QOA

```
Вход: QT: дерево запроса
Выход: strat: стратегия с минимальной стоимостью
begin
    for каждого отношения R_i ∈ QT do
        for каждого пути доступа AP_ij к R_i do
            вычислить cost(AP_ij)
        end for
        best_AP_i ← AP_ij с минимальной стоимостью
    end for
    for каждого порядка (R_i1, R_i2, ..., R_in), где i = 1, ... , n! do
        построить стратегию (. . .((best_AP_i1 ⋈ R_i2) ⋈ R_i3) ⋈ .... ⋈ R_in)
        вычислить стоимость стратегии
    end for
    strat ← стратегия с минимальной стоимостью
    for каждого узла k, где хранится отношение, участвующее в QT do
        LS_k ← локальная стратегия (стратегия, k)
        передать (LS_k, узел k)                    {каждая локальная стратегия
                                                    оптимизируется в узле k}
    end for
end
```

Компромисс между этими двумя методами очевиден. В случае передачи целиком данных передается больше, но сообщений меньше, чем в случае выборки по мере необходимости. Интуитивно понятно, что передавать целиком лучше небольшие отношения. С другой стороны, если отношение велико и избирательность соединения хорошая (сопоставляемых кортежей немного), то релевантные кортежи лучше выбирать по мере необходимости. Оптимизатор не рассматривает все возможные комбинации методов соединения с методами передачи, поскольку некоторые не заслуживают внимания. Например, было бы бесполезно передавать внешнее отношение методом выборки по мере необходимости в алгоритме вложенных циклов, потому что все внешние кортежи придется обработать в любом случае, поэтому передавать отношение следует целиком.

Для соединения внешнего отношения R с внутренним отношением S по атрибуту A имеется четыре стратегии. Далее мы опишем их по очереди и для каждой приведем упрощенную формулу стоимости, в которой *LT* обозначает время локальной обработки (ввода-вывода и процессора), а *CT* – время передачи данных. Для простоты игнорируем стоимость порождения результата.

Для удобства обозначим s среднее число кортежей S, сопоставляемых с одним кортежем R:

$$s = \frac{card(\mathrm{S} \bowtie_{\mathrm{A}} \mathrm{R})}{card(\mathrm{R})}.$$

Стратегия 1. Передать все внешнее отношение в узел внутреннего отношения. В этом случае внешние кортежи можно соединять с S по мере поступления. Поэтому

Total_time = *LT*(выбрать *card*(R) кортежей из R)
+ *CT*(*size*(R))
+ *LT*(выбрать *s* кортежей из S) ∗ *card*(R).

Стратегия 2. Передать все внутреннее отношение в узел внешнего отношения. В этом случае внутренние кортежи нельзя соединять по мере поступления, а нужно сохранять во временном отношении T. Поэтому

Total_time = *LT*(выбрать *card*(S) кортежей из S)
+ *CT*(*size*(S))
+ *LT*(сохранить *card*(S) кортежей в T)
+ *LT*(выбрать *card*(R) кортежей из R)
+ *LT*(выбрать *s* кортежей из T) * *card*(R).

Стратегия 3. По мере необходимости выбирать кортежи из внутреннего отношения для каждого кортежа из внешнего отношения. В данном случае для каждого кортежа в R значение атрибута соединения (A) передается в узел S. Затем выбираются *s* кортежей из S, сопоставленных с этим значением, и передаются в узел R, где соединяются по мере поступления. Поэтому

Total_time = *LT*(выбрать *card*(R) кортежей из R)
+ *CT*(*length*(A)) ∗ *card*(R)
+ *LT*(выбрать *s* кортежей из S) ∗ *card*(R)
+ *CT*(*s* ∗ *length*(S)) ∗ *card*(R).

Стратегия 4. Переместить оба отношения в третий узел и там вычислить соединение. В этом случае внутреннее отношение перемещается первым и сохраняется во временном отношении T. Затем перемещается внешнее отношение, и его кортежи соединяются с T по мере поступления. Таким образом, имеем

Total_time = *LT*(выбрать *card*(S) кортежей из S)
+ *CT*(*size*(S))
+ *LT*(сохранить *card*(S) кортежей в T)
+ *LT*(выбрать *card*(R) кортежей из R)
+ *CT*(*size*(R))
+ *LT*(выбрать *s* кортежей из T) ∗ *card*(R).

Пример 4.18. Рассмотрим запрос, состоящий из соединения внешнего отношения PROJ с внутренним отношением ASG по атрибуту PNO. Предполагается,

что PROJ и ASG хранятся в двух разных узлах и что по атрибуту PNO над ASG построен индекс. Возможные стратегии выполнения этого запроса таковы:

1) передать PROJ целиком в узел ASG;
2) передать ASG целиком в узел PROJ;
3) выбирать кортежи из ASG по мере необходимости для каждого кортежа из PROJ;
4) переместить ASG и PROJ в третий узел.

Алгоритм оптимизации предсказывает полное время для каждой стратегии и выбирает самую дешевую. При условии что после соединения PROJ ⋈ ASG нет никакого оператора, стратегия 4, очевидно, самая дорогая, потому что перемещать приходится оба отношения. Если *size*(PROJ) много больше *size*(ASG), то стратегия 2 минимизирует время передачи данных и, вероятно, является лучшей, если время локальной обработки не слишком велико по сравнению со стратегиями 1 и 3. Заметим, что время локальной обработки в стратегиях 1 и 3, вероятно, гораздо лучше, чем в стратегии 2, потому что используется индекс по атрибуту соединения.

Если стратегия 2 не лучшая, то выбирать приходится между стратегиями 1 и 3. Стоимость локальной обработки в обоих случаях одинакова. Если PROJ велико и сопоставляется немного кортежей ASG, то в стратегии 3 время передачи данных, вероятно, будет наименьшим, поэтому она лучшая. Если же PROJ мало или сопоставляется много кортежей ASG, то лучшей должна быть стратегия 1. ♦

Концептуально этот алгоритм следует рассматривать как исчерпывающий поиск среди всех вариантов, получающихся перестановкой порядка соединения отношений, методов соединения (включая выбор алгоритма соединения), узла результата, пути доступа к внутреннему отношению и режима межузловой передачи. Сложность такого алгоритма комбинаторно зависит от количества участвующих отношений. Но на практике количество вариантов значительно сокращается за счет использования динамического программирования и эвристик. Дерево вариантов строится динамически, и неэффективные ветви отсекаются.

Оценка производительности алгоритма в контексте высокоскоростных сетей (сравнимых с локальными) и среднескоростных глобальных сетей подтверждает заметный вклад затрат на локальную обработку даже в глобальных сетях. В частности, показано, что для распределенного соединения передавать все внутреннее отношение выгоднее, чем метод выборки по мере необходимости.

4.5.3. Гибридный подход

У динамической и статической оптимизаций запроса есть как преимущества, так и недостатки. В динамическом случае оптимизация совмещается с выполнением, поэтому на этапе выполнения принимаются более точные решения. Однако запрос оптимизируется при каждом выполнении. Поэтому такой подход предпочтительнее для ситуативных одноразовых запросов. Статическая оптимизация производится на этапе компиляции, и время аморти-

зируется между несколькими выполнениями запроса. Поэтому критически важна точность модели стоимости, позволяющей предсказывать стоимость потенциальных ПВЗ. Этот подход лучше для запросов внутри хранимых процедур и принят всеми коммерческими СУБД.

Однако даже при наличии хорошей модели стоимости имеется важная проблема, которая мешает точно оценивать стоимость и сравнивать ПВЗ на этапе компиляции. Беда в том, что значения параметров запросов в хранимой процедуре не известны до момента времени выполнения. Рассмотрим, например, предикат выборки `WHERE R.A=$a`, где `$a` – значение параметра. Чтобы оценить мощность выборки, оптимизатор должен опираться на предположение о равномерном распределении значений `A` в `R` и не может воспользоваться гистограммами. Из-за динамического связывания параметра *a* точно оценить избирательность $\sigma_{A=\$a}(R)$ невозможно до этапа выполнения. Поэтому возможны серьезные ошибки оценивания и, как следствие, выбор неоптимального ПВЗ. Помимо неизвестных значений параметров запроса, может случиться, что во время выполнения некоторые узлы недоступны или перегружены, а отношения (или их фрагменты) реплицированы в разных узлах. Поэтому узел и реплику нужно выбирать во время выполнения, чтобы повысить надежность и сбалансировать нагрузку на систему.

Гибридная оптимизация запроса – это попытка сохранить преимущества статической оптимизации, избежав проблем, свойственных неточным оценкам. В основном подход статический, но дополнительные решения могут приниматься на этапе выполнения. Общее решение состоит в том, чтобы порождать *динамические ПВЗ*, включающие тщательно отобранные решения по оптимизации, которые должны приниматься на этапе выполнения, – для этого служат операторы «выбора плана». Оператор выбора плана связывает два или более эквивалентных подпланов ПВЗ, которые невозможно сравнить на этапе компиляции из-за отсутствия информации, важной для оценки стоимости (например, значений параметров). В процессе выполнения оператора выбора плана подпланы сравниваются на основе фактических затрат, и выбирается наилучший. Узлы выбора плана можно вставить в любое место ПВЗ. Этот подход настолько общий, что позволяет включать решения о выборе узла и реплики. Но пространство поиска альтернативных подпланов, связанных операторами выбора плана, оказывается гораздо обширнее, поэтому статические планы могут получиться тяжеловесными, так что время подготовки к выполнению заметно увеличится. Поэтому предложено несколько гибридных методов оптимизации запросов в распределенных системах. По существу, в них используется двухшаговый подход.

1. На этапе компиляции сгенерировать статический план, который описывает порядок операторов и методы доступа, не учитывая, где хранятся отношения.
2. На этапе подготовки сгенерировать план выполнения, выбрав узел и реплику и распределив операторы по узлам.

Пример 4.19. Рассмотрим следующий вопрос, выраженный в терминах реляционной алгебры:

$\sigma(R_1) \bowtie R_2 \bowtie R_3$.

На рис. 4.15 показан двухшаговый план для этого запроса. Статический план показывает порядок реляционных операторов, найденный централизованным оптимизатором запросов. План времени выполнения дополняет статический план выбором узлов и реплик, а также операциями передачи данных между узлами. Например, первая выборка размещена в узле S_1, применяется к копии R_{11} отношения R_1 и передает результат на узел S_3, где он соединяется с R_{23}, и т. д. ♦

Первый шаг может сделать централизованный оптимизатор запросов. Он также может включить операторы выбора плана, чтобы во время подготовки можно было принять во внимание фактические значения параметров и дать точные оценки стоимости. На втором шаге производится выбор узла и реплики, возможно, в дополнение к выполнению операторов выбора плана. Кроме того, на этом же этапе можно сбалансировать нагрузку на систему. Далее в этом разделе мы проиллюстрируем именно второй шаг.

Рассматривается распределенная база данных с набором узлов $S = \{S_1, \ldots, S_n\}$. Запрос Q представлен в виде упорядоченной последовательности подзапросов $Q = \{q_1, \ldots, q_m\}$. Каждый подзапрос q_i – это максимальная единица обработки, которая обращается к одному базовому отношению и взаимодействует с соседними подзапросами. Например, на рис. 4.15 мы видим три подзапроса: по одному для R_1, R_2 и R_3. С каждым узлом S_i ассоциирована нагрузка $load(S_i)$, отражающая текущее количество переданных ему запросов. Нагрузку можно выражать по-разному, например как количество запросов, ограниченных вводом-выводом и процессором. Средняя нагрузка на систему определена формулой

$$Avg\_load(S) = \frac{\sum_{i=1}^{n} load(S_i)}{n}.$$

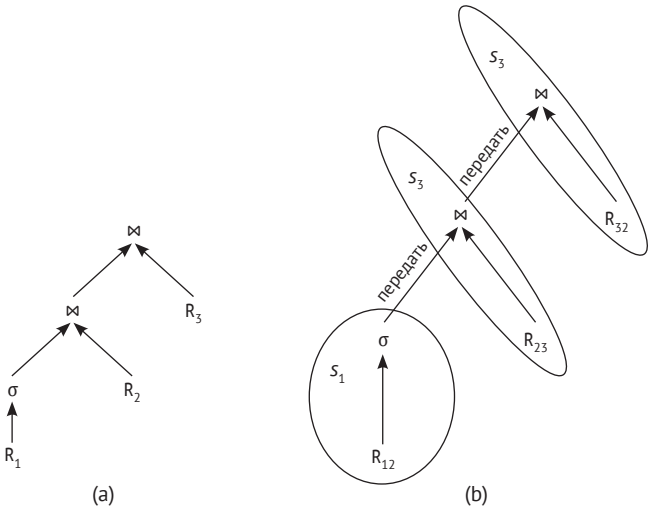


Рис. 4.15 ❖ Двухшаговый план:
(a) статический план; (b) план времени выполнения

Сбалансированность нагрузки на систему для данного распределения подзапросов по узлам можно измерить как дисперсию загруженности узлов, вычислив *коэффициент дисбаланса*:

$$UF(S) = \frac{1}{n}\sum_{i=1}^{n}(load(S_i) - Avg\_load(S))^2.$$

Когда система близка к сбалансированной, коэффициент дисбаланса стремится к 0 (идеальная сбалансированность). Например, при $load(S_1) = 10$, $load(S_2) = 30$ коэффициент дисбаланса системы $\{S_1, S_2\} = 100$, а при $load(S1) = 20$ и $load(S_2) = 20$ он равен 0.

Задачу, решаемую на втором шаге двухшаговой оптимизации запроса, можно формально поставить как следующую задачу размещения подзапросов. Пусть даны:

1) набор узлов $S = \{S_1, \ldots, S_n\}$ и нагрузки на каждый узел;
2) запрос $Q = \{q_1, \ldots, q_m\}$;
3) для каждого подзапроса q_i в Q набор допустимых вариантов размещения в узлах $S_q = \{S_1, \ldots, S_k\}$, где в каждом узле хранится копия отношения, участвующего в q_i.

Требуется найти оптимальное размещение Q на S такое, что

1) $UF(S)$ достигает минимума и
2) полная стоимость передачи данных достигает минимума.

Существует алгоритм, который находит почти оптимальные решения за разумное время. В этом алгоритме (см. алгоритм 4.3), который мы опишем для линейных деревьев соединений, используется несколько эвристик. Первая эвристика (шаг 1) – начать с размещения подзапросов с наименьшей гибкостью размещения, т. е. тех, для которых набор вариантов размещения наименьший. Таким образом, подзапросы, для которых мало узлов-кандидатов, размещаются первыми. Вторая эвристика (шаг 2) – рассматривать узлы с наименьшей нагрузкой и наибольшим выигрышем. Выигрыш узла определяется как количество подзапросов, уже размещенных на нем, и измеряет экономию затрат на передачу данных, получающуюся при размещении подзапроса в этом узле. Наконец, на шаге 3 пересчитывается информация о нагрузке для каждого неразмещенного подзапроса, для которого выбранный узел принадлежит его набору допустимых вариантов размещения.

Пример 4.20. Рассмотрим следующий запрос Q, выраженный в терминах реляционной алгебры:

$$\sigma(\mathtt{R}_1) \bowtie \mathtt{R}_2 \bowtie \mathtt{R}_3 \bowtie \mathtt{R}_4.$$

На рис. 4.16 показано размещение копий 4 отношений на 4 узлах вместе с нагрузками на узел. Предполагается, что Q разложен на четыре подзапроса $Q = \{q_1, q_2, q_3, q_4\}$, где q_1 ассоциирован с $\mathtt{R}_1$, q_2 – с $\mathtt{R}_2$, соединенным с результатом q_1, q_3 – с $\mathtt{R}_3$, соединенным с результатом q_2, и q_4 – с $\mathtt{R}_4$, соединенным с результатом q_3. Алгоритм SQAllocation выполняет 4 итерации. На первой итерации он выбирает подзапрос q_4 с наименьшей гибкостью размещения, размещает его на S_1 и обновляет нагрузку на S_1, делая ее равной 2. На второй итерации следующим выбирается подзапрос q_2 или q_3, поскольку гибкость размещения

для обоих одинакова. Выберем q_2 и предположим, что он размещен на S_2 (но можно было бы разместить и на S_4, потому что нагрузка на него такая же, как на S_2). Нагрузка на S_2 увеличивается до 3. На третьей итерации выбирается следующий подзапрос q_3 и размещается на S_1, который имеет такую же нагрузку, как S_3, но выигрыш 1 (тогда как для S_3 выигрыш равен 0) в результате размещения q_4. Нагрузка на S_1 увеличивается до 3. Наконец, на последней итерации подзапрос q_1 размещается на S_3 или S_4, поскольку нагрузки на них наименьшие. Если бы на второй итерации q_2 был размещен на S_4, а не на S_2, то на четвертой итерации q_1 был бы размещен на S_4, потому что выигрыш равен 1. Это дало бы лучший план выполнения с меньшим объемом передачи данных. Тем самым мы показали, что двухшаговый алгоритм оптимизации все же может пропускать оптимальные планы. ♦

Узлы	Нагрузка	R_1	R_2	R_3	R_4
s_1	1	R_{11}		R_{31}	R_{41}
s_2	2		R_{22}		
s_3	2	R_{13}		R_{33}	
s_4	2	R_{14}	R_{24}		

Рис. 4.16 ❖ Пример размещения данных и нагрузок

Алгоритм 4.3. SQAllocation

Вход: Q: $q_1, \ldots, q_m$
Допустимые варианты размещения: $F_{q1}, \ldots, F_{qm}$
Нагрузки: $load(F_1), \ldots, load(F_m)$
Выход: размещение Q на S
begin
  for *каждого q в Q* **do**
    вычислить($load(F_q)$)
  end for
  while *Q не пуст* **do**
    {выбрать подзапрос a для размещения}
    $a \leftarrow q \in Q$ с наименьшей гибкостью размещения (1)
    {выбрать лучший узел b для a}
    $b \leftarrow f \in F_a$ с наименьшей нагрузкой и лучшим выигрышем (2)
    $Q \leftarrow Q - a$
    {при необходимости пересчитать нагрузки для оставшихся
    допустимых вариантов размещения} (3)
    for *каждого* $q \in Q$, *где* $b \in F_q$ **do**
      вычислить($load(F_q)$)
    end for
  end while
end

Сложность этого алгоритма находится в разумных пределах. Он рассматривает по очереди каждый подзапрос, учитывая все возможные узлы, выби-

рает текущий для размещения и сортирует список оставшихся подзапросов. Следовательно, сложность можно оценить как $\mathcal{O}(\max(m * n, m^2 * \log_2 m))$.

Наконец, алгоритм включает шаг уточнения, чтобы дополнительно оптимизировать обработку соединений и решить, использовать полусоединения или нет. Хотя двухшаговая оптимизация запроса минимизирует передачу данных при данном статическом плане, она может порождать планы времени выполнения, для которых стоимость передачи данных больше, чем в оптимальном плане. Это происходит, потому что на первом шаге игнорируется местоположение данных и его влияние на стоимость коммуникации. Например, рассмотрим план времени выполнения на рис. 4.15 и предположим, что третий подзапрос к R_3 размещен в узле S_1 (а не S_2). В этом случае план, который предусматривает сначала соединение (или декартово произведение) результата выборки R_1 с R_3 в узле S_1, может оказаться лучше, потому что минимизирует объем передачи данных. Решение этой проблемы в том, чтобы реорганизовать план, применив преобразования к дереву операторов на этапе подготовки к выполнению запроса.

4.6. Адаптивная обработка запроса

До сих пор мы предполагали, что процессор распределенных запросов знает достаточно об условиях во время выполнения запроса, чтобы сгенерировать эффективный ПВЗ, и что эти условия остаются неизменными во время выполнения. Это предположение справедливо для запросов с небольшим числом отношений в контролируемой среде. Однако оно не действует в динамично изменяющейся среде с большим количеством отношений и непредсказуемыми условиями во время выполнения.

Пример 4.21. Рассмотрим ПВЗ на рис. 4.17 с отношениями EMP, ASG, PROJ и PAY в узлах S_1, S_2, S_3, S_4 соответственно. Перечеркнутая стрелка означает, что по какой-то причине (например, из-за сбоя) узел S_2 (где хранится ASG) в начале выполнения недоступен. Предположим для простоты, что запрос выполняется в соответствии с итераторной моделью, так что кортежи поступают из самого левого отношения.

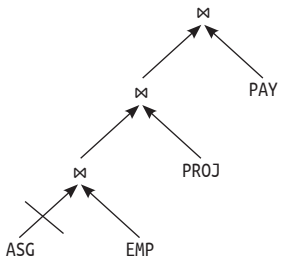


Рис. 4.17 ❖ План выполнения запроса с заблокированным отношением

Из-за недоступности S_2 весь конвейер блокирован в ожидании поступления кортежей из ASG. Однако, реорганизовав план, мы могли бы вычислить другие операторы, пока ждем S_2, например соединение EMP и PAY. ♦

Этот простой пример показывает, что типичный статический план не может справиться с непредсказуемой недоступностью источника данных. Можно привести и более сложные примеры: постоянные запросы, дорогостоящие предикаты и асимметрия данных. Решение заключается в том, чтобы включить в состав обработки запросов какое-то адаптивное поведение, т. е. реализовать *адаптивную обработку запросов*. Это форма динамической обработки запросов, при которой между исполняющей средой и оптимизатором запросов реализован контур обратной связи, позволяющий реагировать на непредвиденные изменения условий во время выполнения. Система обработки запросов называется адаптивной, если она получает информацию от исполняющей среды и итеративно подстраивает свое поведение под эту информацию.

В этом разделе мы сначала представим общий обзор адаптивной обработки запроса. А затем опишем вихревой подход – мощную инфраструктуру для адаптивной обработки.

4.6.1. Процесс адаптивной обработки запросов

По сравнению с традиционной адаптивная обработка запросов добавляет следующие операции: мониторинг, оценку и реакцию. Логично, что эти операции реализованы в системе обработки запросов с помощью датчиков, компонентов оценки и компонентов реакции соответственно. Под мониторингом понимается измерение некоторых параметров среды в течение временного окна и передача их компоненту оценки. Тот анализирует сообщения и с учетом пороговых значений вырабатывает адаптивный план реагирования. Наконец, план реагирования передается компоненту реакции, который применяет его к выполнению запроса.

Типичный адаптивный процесс выполняет компоненты с заданной частотой. Существует компромисс между реактивностью, когда более высокое значение приводит к более быстрой реакции, и накладными расходами адаптивного процесса. В общем случае адаптивный процесс описывается функцией $f_{adapt}(E, T) \rightarrow Ad$, где E – набор отслеживаемых параметров среды, T – набор пороговых значений, а Ad – возможно, пустое множество адаптивных реакций. Элементы E, T и Ad, называемые адаптивными элементами, очевидно, могут меняться в широких пределах в зависимости от приложения. Самыми важными являются отслеживаемые параметры и адаптивные реакции.

4.6.1.1. Отслеживаемые параметры

Мониторинг параметров запроса предполагает размещение датчиков в ключевых местах ПВЗ и определение окон наблюдения, в течение которых датчики собирают информацию. Также требуется специфицировать механизм передачи собранной информации компоненту оценки. Приведем несколько кандидатов на отслеживание.

- Размер памяти. Мониторинг размера доступной памяти, например, позволяет операторам реагировать на нехватку или добавление памяти.
- Скорость поступления данных. Мониторинг изменений скорости поступления данных позволяет процессору запросов делать какую-то полезную работу, пока источник данных заблокирован.
- Актуальная статистика. Статистика базы данных в распределенной среде чаще всего неточна, а то и вовсе недоступна. Мониторинг фактического размера отношений и промежуточных результатов может стать основанием для внесения важных изменений в ПВЗ. Кроме того, можно отказаться от стандартных предположений о том, что избирательности предикатов по разным атрибутам отношения независимы, и вычислить истинную избирательность для разных значений.
- Стоимость выполнения операторов. Мониторинг фактической стоимости выполнения операторов, в т. ч. темпов производства данных, полезен для улучшенного планирования операторов. А мониторинг размера очередей к операторам поможет избежать перегрузки.
- Пропускная способность сети. Мониторинг пропускной способности сети может быть полезен для определения размера блока выборки данных. В сети с низкой пропускной способностью система может отдавать блоки большего размера, чтобы снизить сетевые издержки.

4.6.1.2. Адаптивные реакции

Адаптивные реакции модифицируют способ выполнения запроса в соответствии с решениями, принятыми компонентом оценки. Перечислим некоторые важные адаптивные реакции.

- Изменение порядка: изменяет порядок, в котором запланировано выполнение операторов в ПВЗ. *Реорганизация запроса* (query scrambling) реагирует *изменением плана*, чтобы избежать ожидания заблокированного источника данных во время выполнения запроса. Вихрь (eddy) – это более тонкая реакция, когда операторы планируются на уровне кортежей.
- Замена оператора: заменяет физический оператор эквивалентным. Например, в зависимости от доступной памяти система может выбрать либо соединение методом вложенных циклов, либо соединение хешированием. Замена оператора может также вылиться в изменение плана путем добавления нового оператора, соединяющего промежуточные результаты, порожденные предыдущей адаптивной реакцией. Например, реорганизация запроса может приводить к введению новых операторов для вычисления соединения между результатами реакций на *изменение порядка*.
- Перефрагментация данных: рассматривается возможность динамической фрагментации отношения. Статическое секционирование иногда приводит к несбалансированной нагрузке на узлы. Например, если информация фрагментирована в соответствии с географическим регионом, то скорость доступа может меняться в течение дня из-за различий в часовых поясах.

- Пересчет плана: вычисляет новый ПВЗ, заменяющий неэффективный. При этом оптимизатор учитывает актуальную статистику и информацию о состоянии, собираемую динамически.

4.6.2. Вихревой подход

Вихревой подход – это общая инфраструктура адаптивной обработки запроса к распределенным отношениям. Для простоты будем рассматривать только запросы вида выборка–проекция–соединение (ВПС). Операторы выборки могут включать дорогостоящие предикаты. Процесс генерирования ПВЗ по входному ВПС-запросу начинается созданием дерева операторов для графа соединений G входного запроса. При выборе алгоритмов соединения и методов доступа к отношениям предпочтение отдается адаптивности. ПВЗ можно смоделировать как кортеж $Q = \langle D,P,C\rangle$, где D – множество отношений базы данных, P – множество предикатов запроса вместе с ассоциированными алгоритмами, C – множество ограничений на порядок, которые должны соблюдаться во время выполнения. Заметим, что по G можно построить несколько допустимых деревьев операторов, не нарушающих ограничений из C, если исследовать пространство поиска с разными порядками предикатов. Но во время компиляции нет нужды искать оптимальный ПВЗ. Порядок операторов определяется динамически на уровне кортежей (так называемая *маршрутизация кортежей*). Процесс компиляции ПВЗ дополняется *вихревым оператором* – n-арным оператором, размещенным между отношениями из D и предикатами запроса из P.

Пример 4.22. Рассмотрим запрос с тремя отношениями $Q = (\sigma_p(\mathtt{R}) \bowtie \mathtt{S} \bowtie \mathtt{T})$, где имеются в виду эквисоединения. Предположим, что единственный метод доступа к отношению `T` – доступ по индексу по атрибуту соединения `T.A`, т. е. второе соединение может быть только индексным соединением по `T.A`. Предположим также, что σ_p – дорогостоящий предикат (например, предикат над результатом выполнения программы для значений `T.B`). В этих предположениях ПВЗ определен как $D = \{\mathtt{R}, \mathtt{S}, \mathtt{T}\}$, $P = \{\sigma_p(\mathtt{R}), \mathtt{R} \bowtie \mathtt{S}, \mathtt{S} \bowtie \mathtt{T}\}$, $C = \{\mathtt{S} \prec \mathtt{T}\}$. Ограничение $\prec$ означает, что кортежи `S` сравниваются с кортежами `T` с использованием индекса по `T.A`.

На рис. 4.18 показан ПВЗ, порожденный компиляцией запроса Q с вихревым оператором. Эллипс соответствует физическому оператору (т. е. либо вихревому оператору, либо алгоритму, реализующему предикат $p \in P$). Как обычно, в нижней части плана представлены исходные отношения. В отсутствие метода доступа просмотром доступ к отношению `T` обернут соединением $\mathtt{S} \bowtie \mathtt{T}$, поэтому в виде исходного отношения `T` не присутствует. Стрелки описывают конвейерный поток данных типа производитель–потребитель. Наконец, стрелка, ведущая наружу из вихря, моделирует порождение выходных кортежей. ♦

Вихрь обеспечивает мелкоструктурную адаптивность – динамически решает, как маршрутизировать кортежи через предикаты, следуя политике планирования. Во время выполнения запроса кортежи извлекаются из ис-

ходных отношений и поступают во входной буфер, управляемый вихревым оператором. Если некоторое отношение недоступно, то вихрь просто читает из другого отношения и накапливает кортежи во входном буфере.

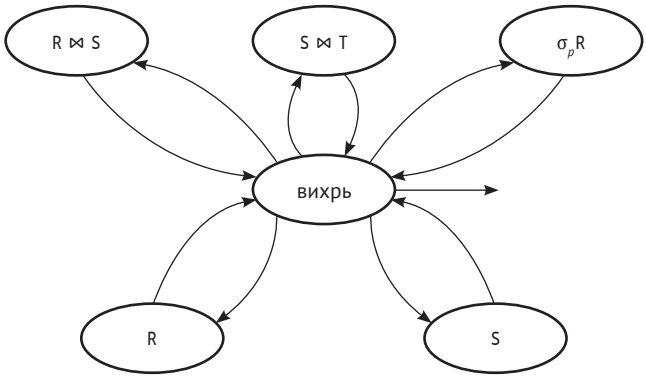


Рис. 4.18 ❖ Выполнение плана с применением вихря

Гибкость выбора текущего доступного исходного отношения достигается путем ослабления фиксированного порядка предикатов в ПВЗ. При вихревом подходе нет фиксированного ПВЗ, а каждый кортеж следует через предикаты собственным путем с учетом ограничений плана и собственной истории вычисления предикатов.

Стратегия маршрутизации на уровне кортежей порождает новую топологию ПВЗ. Вихревой оператор вместе с управляемыми им предикатами образует круговой поток данных, в котором кортежи покидают вихревой оператор, поступая на вычисление предикатам, которые затем возвращают выходные кортежи вихревому оператору. Кортеж выходит из круга в одном из двух случаев: если в результате вычисления предиката оказывается исключенным или если вихревой оператор обнаружил, что кортеж побывал у всех предикатов в списке. Поскольку фиксированного ПВЗ нет, каждый кортеж должен регистрировать набор предикатов, которым он предназначен. Например, на рис. 4.18 кортежи S предназначены обоим предикатам соединения, но не предикату σ_p(R).

4.7. Заключение

В этой главе мы подробно рассказали об обработке запросов в распределенных СУБД. Сначала мы сформулировали задачу распределенной обработки запросов. Основное предположение состоит в том, что входной запрос выражен на языке реляционного исчисления, потому что именно так обстоит дело в большинстве современных распределенных СУБД. Сложность задачи пропорциональна выразительной способности языка запросов и его способности к абстрагированию.

Задача обработки запросов в распределенной среде очень трудна из-за наличия многих элементов. Но ее можно разделить на несколько подзадач, которые по отдельности решить проще. Поэтому мы предложили общую многоуровневую схему описания распределенной обработки запросов. Было выделено четыре основные функции: декомпозиция запроса, локализация данных, распределенная оптимизация и распределенное выполнение. Эти функции последовательно уточняют запрос, добавляя новую информацию о среде обработки.

Затем мы описали локализацию данных, сделав упор на методах редукции и упрощения для четырех видов фрагментации: горизонтальной, вертикальной, производной и гибридной. Запрос, порожденный уровнем локализации данных, хорош в том смысле, что удается избежать худших способов выполнения. Однако на последующих уровнях обычно производятся важные оптимизации, поскольку они добавляют новые сведения о среде обработки.

Далее мы обсудили важную проблему оптимизации, касающуюся порядка соединений в распределенных запросах, затронув альтернативные стратегии соединения, основанные на полусоединении, и определение распределенной модели стоимости.

Мы проиллюстрировали методы соединения и полусоединения в трех основных алгоритмах распределенной оптимизации запросов: динамическом, статическом и гибридном. У статической и динамической оптимизаций те же достоинства и недостатки, что в централизованных системах. Гибридный подход на сегодня считается лучшим в распределенных средах, потому что откладывает принятие важных решений, например выбор реплики и размещение подзапросов в узлах, до момента подготовки к выполнению запроса. Это повышает доступность и улучшает балансировку нагрузки на узлы системы. Мы проиллюстрировали гибридный подход на примере двухшаговой оптимизации запроса, когда сначала генерируется статический план, который задает порядок операторов, как в централизованной системе, а затем генерируется план выполнения на этапе подготовки, и в этот момент выбирается узел результата и реплика, а операторы распределяются по узлам.

Наконец, мы обсудили адаптивную обработку запросов, которая помогает справиться с динамическим поведением локальных СУБД. Проблема решается с помощью динамического подхода, когда оптимизатор запросов взаимодействует с исполняющей средой на этапе выполнения, чтобы своевременно реагировать на непредвиденные изменения условий.

4.8. Библиографические замечания

Существует несколько обзорных статей на тему обработки и оптимизации запросов в контексте реляционной модели. Подробный обзор см. в работе [Graefe 1993].

Итераторная модель выполнения, положенная в основу многих реализаций процессора запросов, была предложена в ходе работы над расширяемой системой вычисления запросов Volcano [Graefe 1994]. В первопроходческой

работе по стоимостной оптимизации запросов [Selinger et al. 1979] впервые предложена модель стоимости, включающая использование статистики базы данных (см. раздел 4.4.2) и стратегию поиска на основе динамического программирования (см. раздел 4.1.1). Для достижения приемлемого компромисса между временем оптимизации и временем выполнения были предложены рандомизированные стратегии, например итеративное улучшение [Swami 1989] и имитация отжига [Ioannidis andWong 1987].

Самый полный обзор распределенной обработки запросов – работа [Kossmann 2000], в которой рассматриваются как распределенные СУБД, так и мультибазовые системы. В этой работе описаны традиционные этапы обработки запросов в централизованных и распределенных системах, а также различные методы распределенной обработки запросов. Распределенные модели стоимости обсуждаются в нескольких работах, в частности [Lohman et al., Khoshafian and Valduriez 1987].

Локализация данных подробно рассматривается в работе [Ceri and Pelagatti 1983] для горизонтально секционированных отношений, которые в ней названы мультиотношениями. Формальные свойства горизонтальной и вертикальной фрагментаций использованы в работе [Ceri et al. 1986], чтобы охарактеризовать соединения фрагментированных отношений.

Теория полусоединений и ее ценность для распределенной обработки запросов рассмотрены в работах [Bernstein and Chiu 1981], [Chiu and Ho 1980] и [Kambayashi et al. 1982]. Подход к распределенной оптимизации запросов на основе полусоединений предложен в работе [Bernstein et al. 1981] для системы SDD-1 ([Wong 1977]). Полные редукторы на базе полусоединений исследуются в работах [Chiu and Ho 1980], [Kambayashi et al. 1982]. Общая задача нахождения полных редукторов является NP-трудной. Однако для цепных запросов существует полиномиальный алгоритм [Chiu and Ho 1980, Ullman 1982]. Стоимость полусоединений можно минимизировать, воспользовавшись битовыми массивами [Valduriez 1982]. Некоторые другие алгоритмы обработки запросов стремятся найти оптимальную комбинацию соединений и полусоединений [Özsoyoglu and Zhou 1987, Wah and Lien 1985].

Динамический подход к распределенной оптимизации запросов впервые предложен в системе Distributed INGRES [Epstein et al. 1978]. В нем используется топология сети (общая или широковещательная сеть) и алгоритм редукции [Wong and Youssefi 1976], который изолирует все неприводимые запросы и запросы с одним отношением путем отделения.

Статический подход к распределенной оптимизации запросов впервые предложен для системы System R* [Selinger and Adiba 1980]. Это одна из первых работ, в которых признается значимость локальной обработки в производительности распределенных запросов. Экспериментальная проверка, предпринятая в работах [Lohman et al. и Mackert and Lohman 1986a,b], подтвердила это важное положение. Метод выборки по мере необходимости, предложенный в R*, назван соединением связыванием в работе [Haas et al. 1997a].

Общий гибридный подход к оптимизации запросов заключается в использовании операторов выбора плана [Cole and Graefe 1994]. Для распределенных систем предложено несколько гибридных подходов на основе двух-

шаговой оптимизации запросов [Carey and Lu 1986, Du et al. 1995, Evrendilek et al. 1997]. Материал раздела 4.5.3 основан на пионерской работе по двухшаговой оптимизации запросов [Carey and Lu 1986]. В статье [Du et al. 1995] предложены эффективные операторы преобразования линейных деревьев соединений (порождаемых на первом шаге) в кустистые деревья, допускающие большую степень параллелизма. В работе [Evrendilek et al. 1997] предложено решение, позволяющее максимизировать межузловой параллелизм на втором шаге.

Обзор адаптивной обработки запросов см. в работах [Hellerstein et al. 2000, Gounaris et al. 2002]. Основополагающей работой по вихревому подходу, которым мы воспользовались для иллюстрации адаптивной обработки запросов в разделе 4.6, является статья [Avnur and Hellerstein 2000]. Из других важных методов адаптивной обработки запросов упомянем реорганизацию запроса [Amsaleg et al. 1996, Urhan et al. 1998], пульсирующие соединения (ripple join) [Haas and Hellerstein 1999b], адаптивное секционирование [Shah et al. 2003] и избирательный подход (cherry picking) [Porto et al. 2003]. Основными обобщениями вихревого подхода являются модули состояний [Raman et al. 2003] и распределенные вихри [Tian and DeWitt 2003].

Упражнения

Задача 4.1. Предположим, что отношение PROJ демонстрационной базы данных горизонтально фрагментировано следующим образом:

$PROJ_1 = \sigma_{PNO \leq "P2"}(PROJ)$;
$PROJ_2 = \sigma_{PNO > "P2"}(PROJ)$.

Преобразуйте следующий запрос в редуцированный запрос на фрагментах:

```
SELECT ENO, PNAME
FROM   PROJ NATURAL JOIN ASG
WHERE  PNO = "P4"
```

Задача 4.2 (*). Предположим, что отношение PROJ горизонтально фрагментировано, как в задаче 4.1, и что отношение ASG горизонтально фрагментировано следующим образом:

$ASG_1 = \sigma_{PNO \leq "P2"}(ASG)$;
$ASG_2 = \sigma_{"P2" < PNO \leq "P3"}(ASG)$;
$ASG_3 = \sigma_{PNO > "P3"}(ASG)$.

Преобразуйте следующий запрос в редуцированный запрос на фрагментах и определите, лучше ли он, чем фрагментарный запрос:

```
SELECT RESP, BUDGET
FROM   ASG NATURAL JOIN PROJ
WHERE  PNAME = "CAD/CAM"
```

Задача 4.3 (**). Предположим, что отношение PROJ горизонтально фрагментировано, как в задаче 4.1. Пусть еще отношение ASG косвенно фрагментировано как

$$ASG_1 = ASG \ltimes_{PNO} PROJ_1;$$
$$ASG_2 = ASG \ltimes_{PNO} PROJ_2,$$

а отношение EMP вертикально фрагментировано как

$$EMP_1 = \prod_{ENO,ENAME}(EMP);$$
$$EMP_2 = \prod_{ENO,TITLE}(EMP).$$

Преобразуйте следующий запрос в редуцированный запрос на фрагментах:

```
SELECT ENAME
FROM   EMP NATURAL JOIN ASG NATURAL JOIN PROJ
WHERE  PNAME = "Instrumentation"
```

Задача 4.4. Рассмотрите граф соединений на рис. 4.11, дополненный следующей информацией: *size*(EMP) = 100, *size*(ASG) = 200, *size*(PROJ) = 300, *size*(EMP $\ltimes$ ASG) = 300, *size*(ASG $\ltimes$ PROJ) = 200. Опишите оптимальную программу соединения, если в качестве целевой функции используется полное время передачи.

Задача 4.5. Рассмотрите граф соединений на рис. 4.11 в тех же предположениях, что в задаче 4.4. Опишите оптимальную программу соединения, которая минимизирует время ответа (учитывайте только передачу данных).

Задача 4.6. Рассмотрите граф соединений на рис. 4.11 и опишите программу (возможно, не оптимальную), которая полностью редуцирует каждое отношение с помощью полусоединений.

Задача 4.7. Рассмотрите граф соединений на рис. 4.11 и фрагментацию на рис. 4.19. Предположим еще, что *size*(EMP $\bowtie$ ASG) = 2000 и *size*(ASG $\bowtie$ PROJ) = 1000. Примените динамический алгоритм распределенной оптимизации запросов из раздела 4.5.1 в двух случаях: общей и широковещательной сети, стремясь минимизировать время передачи данных.

Отношение	Узел 1	Узел 2	Узел 3
EMP	1000	1000	1000
ASG		2000	
PROJ	1000		

Рис. 4.19 ❖ Фрагментация

Задача 4.8 (**). Рассмотрим следующий запрос к нашей базе данных конструкторской компании:

```
SELECT ENAME,SAL
FROM   PAY NATURAL JOIN EMP NATURAL JOIN ASG
```

```
      NATURAL JOIN PROJ
WHERE (BUDGET>200000 OR DUR>24)
AND   (DUR>24 OR PNAME = "CAD/CAM")
```

Предположим, что отношения EMP, ASG, PROJ и PAY хранятся в узлах 1, 2, 3 в соответствии с таблицей на рис. 4.20. Предположим также, что скорость передачи данных между любыми двумя узлами одинакова и что передача данных в 100 раз медленнее, чем обработка данных в любом узле. Наконец, предположим, что *size*(R ⋈ S) = *max*(*size*(R), *size*(S)) для любых двух отношений R и S и что коэффициент избирательности дизъюнктивной выборки равен 0.5. Напишите распределенную программу, которая вычисляет ответ на такой запрос и минимизирует полное время.

Отношение	Узел 1	Узел 2	Узел 3
EMP	2000		
ASG		3000	
PROJ			1000
PAY			500

Рис. 4.20 ❖ Статистика фрагментации

Задача 4.9 (**). В разделе 4.5.3 мы описали алгоритм 4.3 для линейных деревьев соединений. Обобщите этот алгоритм на кустистые деревья соединений. Примените его к кустистому дереву соединений на рис. 4.9, воспользовавшись размещением данных и нагрузками на узлы, показанными на рис. 4.16.

Задача 4.10 (**). Рассмотрим три отношения R(A, B), S(B, D) и T(D, E) и запрос $Q(\sigma_p^1(R) \bowtie_1 S \bowtie_2 T)$, где $\bowtie_1$ и $\bowtie_2$ – естественные соединения. Предположим, что над S построен индекс по атрибуту B, а над T – индекс по атрибуту D. Предположим также, что σ_p^1 – дорогостоящий предикат (т. е. предикат над результатами работы программы, применяемой к значениям R.A). Воспользовавшись вихревым подходом к адаптивной обработке запроса, решите следующие задачи.

- **a)** Предложить множество *C* ограничений на *Q* для порождения основанного на вихре ПВЗ.
- **b)** Нарисовать граф соединений *G* для *Q*.
- **c)** Используя *C* и *G*, предложить основанный на вихре ПВЗ.
- **d)** Предложить другой ПВЗ, в котором используются модули состояний. Обсудить преимущества использования модулей состояний в этом ПВЗ.

Задача 4.11 (**). Предложите структуру данных для хранения кортежей в буфере вихревого оператора, которая позволила бы быстро выбирать для вычисления следующий кортеж согласно заданным пользователем предпочтениям, например чтобы быстро получить первые результаты.

Глава 5

Распределенная обработка транзакций

Понятие *транзакции* в базах данных используется для обозначения элементарной единицы согласованных и надежных вычислений. Таким образом, после того как стратегия выполнения запроса определена и он представлен в виде набора примитивных операций базы данных, эти операции выполняются как транзакции. Транзакции гарантируют согласованность и долговечность изменений базы данных в условиях конкурентного доступа к одним и тем же элементам данных (при этом не более одной операции доступа может быть обновлением) и возможности отказов.

Термины *согласованный* и *надежный* в определении транзакции следует определить более точно. Мы различаем *согласованность базы данных* и *согласованность транзакции*.

Говорят, что база данных находится в *согласованном состоянии*, если соблюдены все определенные в ней ограничения согласованности (целостности) (см. главу 3). Состояние изменяется в результате операций модификации, вставки и удаления (в совокупности они называются *обновлениями*). Разумеется, мы хотим, чтобы база данных никогда не переходила в несогласованное состояние. Заметим, что база данных может быть (и обычно бывает) временно несогласованной на протяжении выполнения транзакции. Важно, что по завершении транзакции она снова оказывается в согласованном состоянии (рис. 5.1).

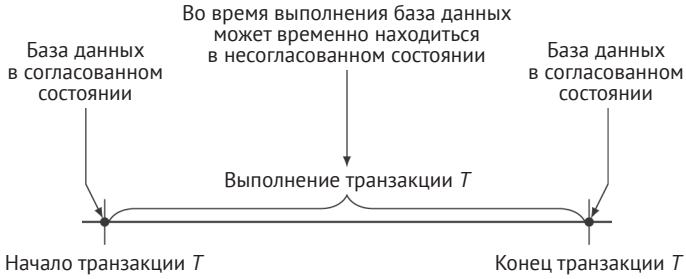


Рис. 5.1 ❖ Транзакционная модель

С другой стороны, под согласованностью транзакции понимаются операции конкурентных транзакций. Мы хотели бы, чтобы база данных оставалась в согласованном состоянии даже в присутствии нескольких пользовательских запросов, конкурентно обращающихся (для чтения или обновления) к базе данных.

Под надежностью понимается *устойчивость* системы к различного рода отказам и ее способность *восстанавливаться* после них. Устойчивая система может функционировать и обслуживать пользователей даже в случае отказа. Говорят, что СУБД допускает восстановление, если она может перейти в согласованное состояние (откатившись назад к предыдущему согласованному состоянию или продвинувшись в новое согласованное состояние) после различных отказов.

Управление транзакциями должно обеспечить неукоснительное нахождение базы данных в согласованном состоянии даже при наличии конкурентного доступа и отказов. Проблемы управления конкурентными транзакциями в централизованных СУБД хорошо известны и описаны во многих учебниках. В этой главе мы исследуем эти проблемы в контексте распределенных СУБД, уделив особое внимание распределенному управлению конкурентностью и распределенной надежности и восстановлению. Предполагается, что читатель знаком с основами управления транзакциями и методами, изучаемыми в курсах и книгах по базам данных. В разделе 5.1 мы кратко напомним основные положения. В этой главе мы игнорируем проблемы, связанные с репликацией данных, этой теме посвящена следующая глава. Принято выделять две категории СУБД: для оперативной обработки транзакций (On-Line Transaction Processing – OLTP) и для оперативной аналитической обработки (On-Line Analytical Processing – OLAP). Приложения для *оперативной обработки транзакций*, например системы бронирования авиабилетов или банковские системы, ориентированы на высокую пропускную способность транзакций. Для них важны развитые средства контроля и доступности данных, высокая пропускная способность при наличии многих пользователей и предсказуемое малое время ответа. С другой стороны, приложения для оперативной аналитической обработки, например анализ тенденций или прогнозирование, должны анализировать исторические сводные данные, поступающие из ряда оперативных баз данных. В них нередки сложные запросы к потенциально очень большим таблицам. В большинстве OLAP-приложений не нужны самые последние версии данных и нет нужды обращаться к актуальным оперативным данным. В этой главе мы будем заниматься OLTP-системами, а OLAP-системы отложим до главы 7.

Эта глава построена следующим образом. В разделе 5.1 мы даем краткое введение в основную терминологию, используемую в этой главе, и возвращаемся к архитектурной модели, определенной в главе 1, чтобы подчеркнуть изменения, которые необходимы для поддержки управления транзакциями. В разделе 5.2 приводится углубленное рассмотрение распределенных методов управления конкурентностью, основанных на сериализуемости. В разделе 5.3 рассматривается управление конкурентностью на основе изоляции снимков. В разделе 5.4 обсуждаются методы распределенного обеспечения

надежности с упором на протоколы распределенной фиксации, завершения и восстановления.

5.1. Основные понятия и терминология

В этом разделе наша цель – дать очень краткое введение в понятия и терминологию, которые понадобятся в дальнейшем, не углубляясь в фундаментальные концепции. Мы также обсудим, как нужно доработать архитектуру системы, чтобы включить в нее транзакции.

Как уже было сказано, транзакция – это единица согласованных и надежных вычислений. Каждая транзакция начинается командой `Begin_transaction`, включает ряд операций `Read` и `Write` и заканчивается командой `Commit` или `Abort`. Команда `Commit` гарантирует, что обновления, совершенные транзакцией в базе данных, с этого момента станут постоянными, а команда `Abort` отменяет действия транзакции, так что, с точки зрения базы данных, эта транзакция вообще никогда не выполнялась. Каждая транзакция характеризуется *множеством чтения* (read set – *RS*), которое включает прочитанные ей элементы данных, и *множеством записи* (write set – *WS*), которое включает записанные элементы данных. Эти множества необязательно являются непересекающимися. Объединение множеств чтения и записи транзакции составляет ее *базовое множество* (base set) ($BS = RS \cup WS$).

Транзакционные службы типичной СУБД обладают свойствами ACID:

1) *атомарность* (Atomicity) гарантирует, что транзакция выполняется атомарно, т. е. либо все составляющие ее операции отражены в базе данных, либо не отражена ни одна;
2) *согласованность* (Consistency) относится к правильности выполнения транзакции (т. е. код транзакции корректен, и после применения к согласованной базе данных транзакция оставляет ее в согласованном состоянии);
3) *изоляция* (Isolation) означает, что результаты одной транзакции не видны никакой другой, пока эта транзакция не зафиксирована – так обеспечивается корректность конкурентных транзакций (т. е. конкурентное выполнение транзакций не нарушает согласованность базы данных);
4) *долговечность* (Durability) означает, что результат зафиксированной транзакции остается в базе данных и не исчезнет даже в случае сбоя системы.

Алгоритмы управления конкурентностью, обсуждаемые в разделе 5.2, гарантируют свойство изоляции, т. е. конкурентные транзакции видят базу данных в согласованном состоянии и оставляют ее согласованной. А меры обеспечения надежности, рассматриваемые в разделе 5.4, гарантируют атомарность и долговечность. Согласованность в том смысле, что транзакция не делает ничего недопустимого с базой данных, обычно обеспечивается контролем целостности, обсуждаемым в главе 3.

Алгоритмы управления конкурентностью реализуют концепцию «корректного конкурентного выполнения». Самая распространенная концепция корректности – *сериализуемость*, когда требуется, чтобы история, порожденная в результате конкурентного выполнения транзакций, была эквивалентна некоторой последовательной истории (т. е. была результатом последовательного выполнения тех же транзакций). Учитывая, что транзакция переводит одно согласованное состояние базы данных в другое, тоже согласованное, любое последовательное выполнение, по определению, корректно; если конкурентное выполнение приводит к истории, эквивалентной одной из этих последовательных историй, то оно тоже должно быть корректно. В разделе 5.3 мы введем ослабленную концепцию корректности – *изоляцию снимков* (snapshot isolation – SI). Алгоритмы управления конкурентностью обычно ставят целью очень эффективно обеспечить различные уровни изоляции конкурентных транзакций.

После завершения транзакции ее результаты следует сделать постоянными. Для этого необходимы *журналы транзакций*, в которых записываются все действия, произведенные транзакцией. Протоколы фиксации гарантируют, что обновления базы данных, а также журналы сохраняются в постоянной памяти, где находятся в безопасности. С другой стороны, протоколы отмены пользуются журналами транзакций, чтобы стереть все следы отмененной транзакции из базы данных. В процессе восстановления системы после сбоя журналы нужны для приведения базы данных в согласованное состояние.

Если мы хотим ввести в СУБД транзакции, помимо одних лишь запросов чтения, то придется вернуться к архитектурной модели, описанной в главе 1, расширить роль монитора распределенного выполнения.

Монитор распределенного выполнения состоит из двух модулей: *диспетчера транзакций* (ДТ) и *планировщика* (ПЛ). Диспетчер транзакций отвечает за координацию выполнения операций базы данных от имени приложения, а планировщик – за реализацию конкретного алгоритма управления конкурентностью с целью синхронизации доступа к базе данных.

Третий компонент, участвующий в управлении распределенными транзакциями, – локальный диспетчер восстановления (ЛДВ), находящийся в каждом узле. Его функция – реализация локальных процедур, позволяющих привести локальную базу данных в согласованное состояние после сбоя.

Каждая транзакция начинается в каком-то одном узле, который мы будем называть *инициирующим узлом*. Выполнение операций с базой данных, составляющих транзакцию, координируется ДМ, который находится в инициирующем узле. Этот ДТ мы будем называть *координатором*, или *координирующим ДТ*.

Диспетчер транзакций реализует интерфейс между прикладными программами и вышеупомянутыми командами транзакций: `Begin_transaction`, `Read`, `Write`, `Commit` и `Abort`. Обработка этих команд в нереплицированной распределенной СУБД обсуждается ниже на абстрактном уровне. Для простоты ограничимся интерфейсом с ДТ, его детали будут описаны в следующих разделах:

1) `Begin_transaction`. Команда сообщает координирующему ДТ о том, что началась новая транзакция. ДТ запоминает имя транзакции, иниции-

ровавшее приложение и прочую информацию в журнале в основной памяти (он называется *непостоянным* журналом);

2) `Read`. Если элемент данных хранится локально, то его значение читается и возвращается транзакции. В противном случае координирующий ДТ определяет, где хранится элемент данных, и запрашивает его значение (предварительно приняв меры по управлению конкурентностью). Узел, на котором прочитаны данные, помещает запись в свой непостоянный журнал;
3) `Write`. Если элемент данных хранится локально, то его значение обновляется (в координации с процессором данных). В противном случае координирующий ДТ определяет, где хранится элемент данных, и запрашивает его обновление (предварительно приняв меры по управлению конкурентностью). Узел, выполнивший запись, помещает запись в свой непостоянный журнал;
4) `Commit`. ДТ координирует узлы, участвующие в обновлении элементов данных от имени этой транзакции, так чтобы сделать обновления, произведенные в каждом узле, долговечными. Протокол WAL служит для перемещения записей из непостоянного журнала на диск (в так называемый *постоянный журнал*);
5) `Abort`. ДТ гарантирует, что от действий, произведенных транзакцией, не осталось никаких следов в базах данных, где были выполнены обновления. Журнал используется для выполнения протокола отмены (отката).

Для предоставления этих служб ДТ может взаимодействовать с планировщиками и процессорами данных в своем узле или в других узлах. Эта конфигурация показана на рис. 5.2.

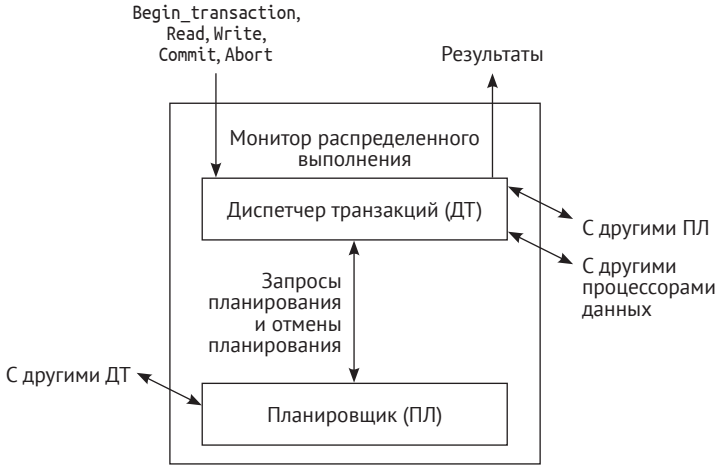


Рис. 5.2 ❖ Подробная модель монитора распределенного выполнения

В главе 1 мы отмечали, что приведенная там архитектурная модель является абстракцией, полезной лишь в педагогических целях. Она позволяет

разделить многие вопросы управления транзакциями и обсуждать их независимо друг от друга. В разделе 5.2 при обсуждении алгоритма планирования мы сосредоточимся на интерфейсах между ДТ и ПЛ, а также между ПЛ и процессором данных. В разделе 5.4 рассмотрим стратегии выполнения команд фиксации и отмены в распределенной среде, а также алгоритмы, которые должен реализовать диспетчер восстановления. В главе мы распространим это обсуждение на реплицированные базы данных. Следует отметить, что описанная нами вычислительная модель не единственная. Были предложены и другие модели, например использование частного рабочего пространства для каждой транзакции.

5.2. Распределенное управление конкурентностью

Как уже было сказано, алгоритм управления конкурентностью гарантирует конкретный уровень изоляции. В этой главе мы в основном сосредоточимся на сериализуемости конкурентных транзакций. Теория сериализуемости непосредственно обобщается на распределенные базы данных. История выполнения транзакций в каждом узле называется *локальной историей*. Если база данных не реплицирована и каждая локальная история сериализуема, то их объединение (*глобальная история*) тоже сериализуемо, при условии что локальные порядки сериализации одинаковы.

Пример 5.1. Приведем очень простой пример, иллюстрирующий это положение. Рассмотрим два банковских счета, x (хранится в узле 1) и y (хранится в узле 2), а также две транзакции: T_1 – перевод 100 долларов с x на y, T_2 – чтение остатков на счетах x и y:

T_1: `Read`(x)
$x \leftarrow x - 100$
`Write`(x)
`Read`(y)
$y \leftarrow y + 100$
`Write`(y)
`Commit`

T_2: `Read`(x)
`Read`(y)
`Commit`

Очевидно, что обе транзакции должны работать в обоих узлах. Рассмотрим следующие две истории, которые могли бы быть созданы локально в обоих узлах (H_i – история в узле i, а R_j и W_j – соответственно операции `Read` и `Write` в составе транзакции T_j):

$H_1 = \{\mathtt{R}_1(x), W_1(x), \mathtt{R}_2(x)\}$;
$H_2 = \{\mathtt{R}_1(y), W_1(y), \mathtt{R}_2(y)\}$.

Обе истории сериализуемые, на самом деле они даже последовательные. Поэтому каждая представляет правильный порядок выполнения. Более того, порядок сериализации в обеих историях одинаков: $T_1 \rightarrow T_2$. Следовательно,

результирующая глобальная история тоже сериализуема, и порядок сериализации в ней $T_1 \rightarrow T_2$.

Однако если бы истории в двух узлах были такими, как показано ниже, то возникла бы проблема:

$H_1' = \{R_1(x), W_1(x), R_2(x)\}$;
$H_2' = \{R_2(y), R_1(y), W_1(y)\}$.

Хотя локальные истории по-прежнему сериализуемые, порядки сериализации различны: в H_1' T_1 предшествует T_2 ($T_1 \rightarrow T_2$), а в H_2' T_2 предшествует T_1 ($T_2 \rightarrow T_1$). Поэтому глобальной сериализуемой истории существовать не может. ♦

Протоколы управления конкурентностью отвечают за изоляцию. Задача протокола – обеспечить конкретный уровень изоляции, например: сериализуемость, изоляция снимков или зафиксированное чтение. Существуют различные аспекты управления конкурентностью. Первый – это, конечно, уровень (или уровни) изоляции, являющий(е)ся целью алгоритма. Второй – запрещает ли протокол нарушение изоляции (пессимистический) или разрешает нарушение, но затем отменяет одну из конфликтующих транзакций, чтобы сохранить уровень изоляции в неприкосновенности (оптимистический).

Третий аспект – как именно сериализуются транзакции. Возможна сериализация в зависимости от порядка конфликтующих операций доступа или в предопределенном *порядке временных меток*. Первый случай соответствует алгоритмам блокировки, когда транзакции сериализуются в том порядке, в каком пытаются захватить конфликтующие блокировки. Второй случай соответствует алгоритмам, которые упорядочивают транзакции в соответствии с временной меткой. Временная метка может назначаться в начале транзакции (время начала), если транзакция пессимистическая, или непосредственно перед фиксацией (время фиксации), если оптимистическая. Четвертый аспект, который мы рассмотрим, – обслуживание обновлений. Первый вариант – хранить одну версию данных (это возможно в пессимистических алгоритмах), второй – хранить несколько версий данных. Последний вариант необходим в оптимистических алгоритмах, но встречается и в некоторых пессимистических для целей восстановления (как правило, хранятся две версии: последняя зафиксированная и текущая незафиксированная). Репликацию мы будем обсуждать в следующей главе.

Большинство комбинаций этих аспектов изучено. Далее в этом разделе мы рассмотрим эталонные методы, применяемые в пессимистических алгоритмах – блокировку (раздел 5.2.1) и упорядочение по временным меткам (раздел 5.2.2) – и в оптимистических алгоритмах (раздел 5.2.4). Понимая эти методы, читатель сможет разобраться и в более сложных алгоритмах.

5.2.1. Алгоритмы на основе блокировки

Алгоритмы управления конкурентностью на основе блокировки предотвращают нарушение изоляции, поддерживая «блокировку» для каждой защищаемой единицы и требуя, чтобы любая операция захватывала блокировку на элемент данных – в режиме чтения (разделяемом) или в режиме запи-

си (монопольном), – прежде чем сможет получить к нему доступ. Решение о том, разрешить ли операцию доступа, принимается в зависимости от совместимости режимов блокировки – блокировка чтения совместима с другой блокировкой чтения, а блокировка записи не совместима ни с какой другой блокировкой. Система управляет блокировками, применяя алгоритм двухфазной блокировки (2PL). Фундаментальное решение в распределенных алгоритмах управления конкурентностью на основе блокировок – где и как хранить блокировки (обычно это место называется *таблицей блокировок*). В следующих разделах описываются алгоритмы, принимающие различные решения в этом отношении.

5.2.1.1. Централизованный алгоритм 2PL

Алгоритм 2PL легко обобщается на распределенные СУБД – нужно лишь делегировать ответственность за управление блокировками одному узлу. Это означает, что только в одном узле имеется диспетчер блокировок, а диспетчеры транзакций в других узлах обращаются к нему для захвата блокировок. Этот подход еще называют алгоритмом 2PL с *главным узлом*.

Взаимодействие между узлами в процессе выполнения транзакции согласно централизованному алгоритму 2PL (C2PL) изображено на рис. 5.3, где показан порядок сообщений. Это взаимодействие имеет место между *координирующим* ДТ, диспетчером блокировок в центральном узле и процессорами данных (ПД) в других участвующих узлах. Участвующими считаются узлы, где хранятся данные, над которыми производятся операции.

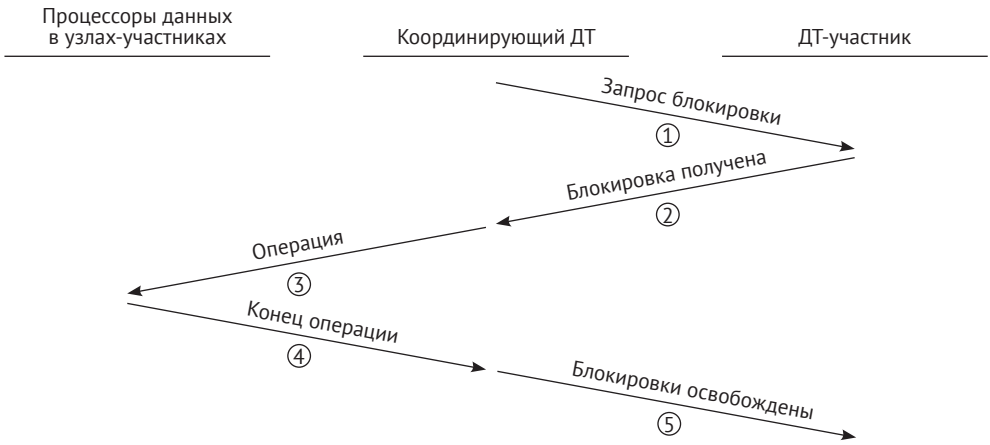


Рис. 5.3 ❖ Коммуникационная структура централизованного 2PL

Централизованный алгоритм 2PL управления транзакциями (C2PL-TM), включающий эти изменения на очень высоком уровне, описан в алгоритме 5.1, а централизованный алгоритм 2PL управления блокировками (C2PL-LM) – в алгоритме 5.2. Сильно упрощенный алгоритм процессора данных (ПД) описан в алгоритме 5.3, который будет значительно переработан при обсуждении надежности в разделе 5.4.

Алгоритм 5.1. Централизованный диспетчер транзакций 2PL (C2PL-TM)

```
Вход: msg: сообщение
begin
    repeat
        ждать msg
        switch msg do
            case транзакционная операция do
                обозначим операцию op
                if op.Type = BT then ПД(op)        {вызвать ПД, передав операцию}
                else C2PL-LM(op)                   {вызвать ДБ, передав операцию}
            end case
            case ответ диспетчера блокировок do    {запрос блокировки одобрен
                                                    или блокировки освобождены}
                if запрос блокировки одобрен then
                    найти узел, в котором хранится запрошенный элемент
                    данных (скажем, H_i)
                ПД_Si(op)                          {вызвать ПД в узле S_i, передав операцию}
                else                               {иначе сообщение об освобождении}
                    проинформировать пользователя о завершении транзакции
                end if
            end case
            case ответ процессора данных do        {сообщение о завершении
                                                    операции}
                switch операция транзакции do
                    обозначим операцию op
                    case R do
                        вернуть приложению op.val (значение элемента данных)
                    end case
                    case W do
                        проинформировать приложение о завершении записи
                    end case
                    case C do
                        if от всех участников получено сообщение
                            о фиксации then
                            проинформировать приложение об успешном
                            завершении транзакции
                            C2PL-LM(op)  {необходимо освободить блокировки}
                        else                       {ждать сообщений о фиксации от всех}
                            запротоколировать приход сообщения о фиксации
                        end if
                    end case
                    case A do
                        проинформировать приложение об отмене
                        C2PL-LM(op)      {необходимо освободить блокировки}
                    end case
                end switch
            end case
        end switch
    until бесконечно
end
```

Алгоритм 5.2. Централизованный диспетчер блокировок 2PL (C2PL-LM)

```
Вход: op : Op
begin
    switch op.Type do
        case R или W do          {запрос блокировки; смотрим, можно ли одобрить}
            найти такую единицу блокировки lu, что op.arg ⊆ lu
            if lu не заблокирована или режим блокировки lu совместим с op.Type then
                установить блокировку на lu в подходящем режиме от имени
                транзакции op.tid
                отправить «Блокировка одобрена» координирующему ДТ
                транзакции
            else
                поместить op в очередь к lu
            end if
        end case
        case C или A do                  {необходимо освободить блокировки}
            foreach единицы блокировки lu, удерживаемой транзакцией do
                освободить блокировку на lu, удерживаемую транзакцией
                if в очереди к lu есть операции then
                    взять первую операцию O из очереди
                    установить блокировку на lu от имени O
                    отправить «Блокировка одобрена» координирующему ДТ
                    транзакции O.tid
                end if
            end foreach
            отправить «Блокировки освобождены» координирующему ДТ
            транзакции
        end case
    end switch
end
```

В этих алгоритмах используется пятерка, описывающая выполняемую операцию: *Op* : ⟨*Type* = {*BT*, *R*, *W*, *A*, *C*}, *arg* : элемент данных, *val* : значение, *tid* : идентификатор транзакции, *res* : результат⟩. Поле *o.Type* ∈ {*BT*, *R*, *W*, *A*, *C*} задает тип операции *o* : *Op*, где *BT* = `Begin_transaction`, *R* = `Read`, *W* = `Write`, *A* = `Abort`, *C* = `Commit`; *arg* – это элемент данных, к которому операция обращается (читает или записывает, для прочих операций это поле равно null); *val* – прочитанное или подлежащее записи значение элемента *arg* (для прочих операций это поле равно null); *tid* – транзакция, в состав которой входит операция (строго говоря, идентификатор этой транзакции), а *res* – код завершения операций, запрошенных у ПД, он важен для работы алгоритмов надежности.

Алгоритм диспетчера транзакций (C2PL-TM) написан как процесс, который работает вечно и ждет сообщения от приложения (содержащего операцию транзакции), или от диспетчера блокировок, или от процессора данных. Алгоритмы диспетчера блокировок (C2PL-LM) и процессора данных (DP) написаны как процедуры, вызываемые по мере необходимости. Поскольку алгоритмы находятся на высоком уровне абстракции, это не очень важно, но, конечно, фактические реализации могут сильно различаться.

Алгоритм 5.3. Процессор данных (DP)

```
Вход: op : Op
begin
    switch op.Type do                                  {ветвление по типу операции}
        case BT do                                     {детали обсуждаются в разделе 5.4}
            служебные действия
        end case
        case R do
            op.res ← READ(op.arg);                     {операция READ}
            op.res ← «Чтение завершено»
        end case
        case W do    {операция WRITE записи значения val в элемент данных arg}
            WRITE(op.arg, op.val)
            op.res ← «Запись завершена»
        end case
        case C do
            COMMIT ;                                   {выполнить COMMIT }
            op.res ← «Фиксация завершена»
        end case
        case A do
            ABORT ;                                    {выполнить ABORT }
            op.res ← «Отмена завершена»
        end case
    end switch
    return op
end
```

Алгоритмы C2PL часто критикуют за то, что центральный узел может быстро стать узким местом. Кроме того, система может оказаться менее надежной, потому что отказ или недоступность центрального узла вызовет сбои во всей системе.

5.2.1.2. Распределенный 2PL

Распределенный алгоритм 2PL (D2PL) требует наличия диспетчеров блокировок в каждом узле. Взаимодействие между исполняющими транзакцию узлами по распределенному протоколу 2PL изображено на рис. 5.4.

Распределенный 2PL алгоритм управления транзакциями похож на C2PL-TM, но с двумя важными модификациями. Сообщения, которые в алгоритме C2PL-TM посылаются диспетчеру блокировок в центральном узле, в D2PL-TM посылаются диспетчерам блокировок во всех участвующих узлах. Второе отличие – то, что процессорам данных операции передаются не координирующим диспетчером транзакций, а участвующими диспетчерами блокировок. Это означает, что координирующий диспетчер транзакций не ждет сообщения «запрос блокировки одобрен». И еще один момент относительно рис. 5.4. Участвующие процессоры данных посылают сообщения «операция завершена» координирующим ДТ. Альтернатива – посылать сообщения собственным диспетчерам блокировок, которые могут в ответ освободить бло-

кировки и уведомить координирующий ДТ. Мы решили остановиться на первом варианте, поскольку в нем используется алгоритм ДБ, идентичный уже описанному строгому 2PL-диспетчеру блокировок, а обсуждение протокола фиксации упрощается (см. раздел 5.4). Благодаря этому сходству мы приводим здесь распределенные алгоритмы ДТ и ДБ. Распределенные алгоритмы 2PL использовались в системах System R* и NonStop SQL.

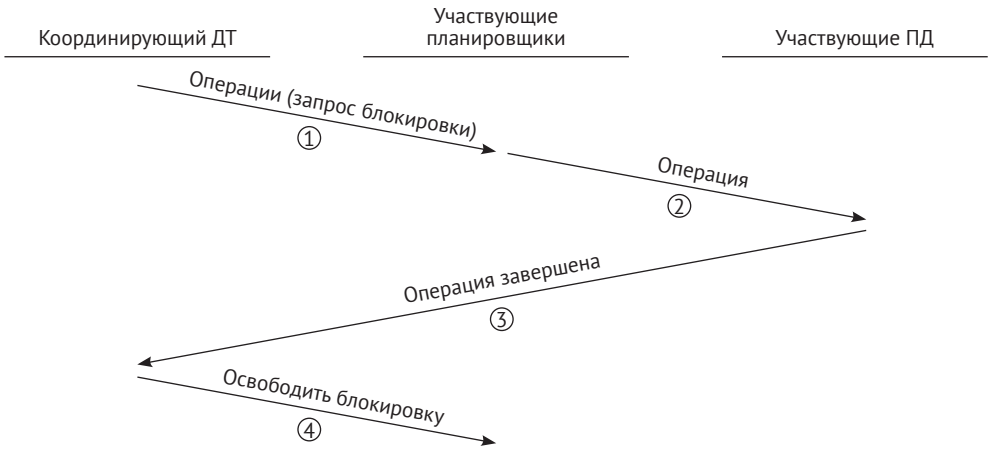


Рис. 5.4 ❖ Структура взаимодействия в распределенном алгоритме 2PL

5.2.1.3. Управление распределенными взаимоблокировками

Алгоритмы управления конкурентностью на основе блокировок могут приводить к взаимоблокировкам; в случае распределенных СУБД взаимоблокировки могут быть распределенными (или глобальными), поскольку транзакции, выполняемые в разных узлах, могут ждать друг друга. Обнаружение и разрешение взаимоблокировок – самый распространенный подход к управлению взаимоблокировками в распределенной среде. Для обнаружения взаимоблокировок полезен граф ожидания (ГО); это ориентированный граф, вершинами которого являются активные транзакции, а из T_i в T_j ведет ребро, если операция в T_i ожидает доступа к элементу данных, заблокированному в несовместимом режиме операцией в T_j. Однако в распределенной среде построения ГО усложняется из-за распределенного выполнения транзакций. Поэтому недостаточно построить в каждом узле *локальный граф ожидания* (ЛГО) и проверить его; необходимо также построить *глобальный граф ожидания* (ГГО), являющийся объединением ЛГО, и проверить наличие в нем циклов.

Пример 5.2. Рассмотрим четыре транзакции T_1, T_2, T_3, T_4, связанные следующими отношениями ожидания: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$. Для случая, когда T_1 и T_2 работают в узле 1, а T_3 и T_4 – в узле 2, ЛГО для обоих узлов показаны на рис. 5.5а. Заметим, что обнаружить взаимоблокировку невозможно путем независимого исследования двух ЛГО, потому что взаимоблокировка гло-

бальная. Но ее легко обнаружить, рассмотрев ГГО – межузловое ожидание показано штриховыми линиями (рис. 5.5b). ♦

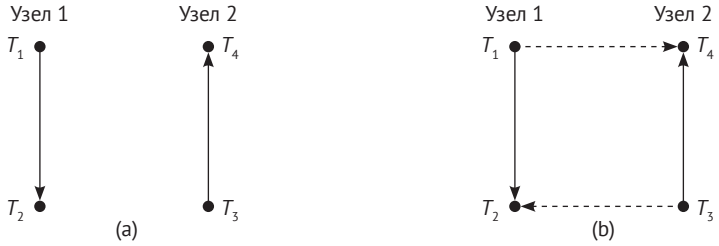


Рис. 5.5 ❖ Различие между ЛГО и ГГО

Алгоритмы различаются тем, как работают с ГГО. Существует три главных метода обнаружения распределенных взаимоблокировок: *централизованное*, *распределенное* и *иерархическое*. Обсудим их.

Централизованное обнаружение взаимоблокировок

В этом случае один узел назначается детектором взаимоблокировок для всей системы. Периодически каждый диспетчер блокировок передает свой ЛГО детектору, а тот строит ГГО и ищет в нем циклы. Диспетчеры блокировок должны передавать только изменения своих графов (т. е. вновь созданные или удаленные ребра). Интервал времени между последовательными передачами – параметры системы: чем он меньше, тем меньше существуют необнаруженные взаимоблокировки, но при этом возрастают накладные расходы на передачу данных и обнаружение.

Централизованное обнаружение взаимоблокировок – простой алгоритм, который является вполне естественным выбором, если в качестве алгоритма управления конкурентностью используется централизованный 2PL. Однако следует принимать во внимание уязвимость к сбоям и высокие накладные расходы на передачу данных.

Иерархическое обнаружение взаимоблокировок

Альтернативой централизованному обнаружению взаимоблокировок является построение иерархии детекторов (рис. 5.6). Взаимоблокировки, локальные для одного узла, обнаруживаются на этом узле с помощью ЛГО. Каждый узел также посылает свой ЛГО детектору взаимоблокировок следующего уровня. Таким образом, распределенные взаимоблокировки с участием двух и более узлов обнаруживаются детектором самого нижнего уровня, контролирующим все эти узлы. Например, взаимоблокировка в узле 1 была бы обнаружена локальным детектором взаимоблокировок (ДВ) в узле 1 (он обозначается $ДВ_{21}$ – 2 обозначает уровень 2, а 1 узел 1). Но если во взаимоблокировке участвуют узлы 1 и 2, то ее обнаружит $ДВ_{11}$. Наконец, если во взаимоблокировке участвуют узлы 1 и 4, то ее обнаружит $ДВ_{0x}$, где x – любое из чисел 1, 2, 3, 4.

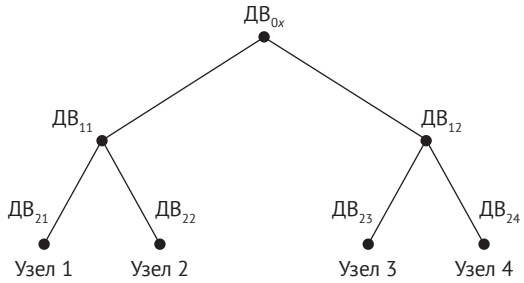


Рис. 5.6 ❖ Иерархическое обнаружение взаимоблокировок

Метод иерархического обнаружения взаимоблокировок уменьшает зависимость от центрального узла, а значит, и затраты на передачу данных. Однако его гораздо труднее реализовать, при этом потребуются нетривиальные модификации диспетчеров блокировок и транзакций.

Распределенное обнаружение взаимоблокировок

В этом случае ответственность за обнаружение взаимоблокировок делегируется отдельным узлам. Как и при иерархическом обнаружении взаимоблокировок, имеются локальные детекторы в каждом узле, которые обмениваются ЛГО между собой (на самом деле передаются только потенциальные циклы, приводящие к взаимоблокировкам). Из различных алгоритмов распределенного обнаружения взаимоблокировок наиболее широко известен тот, что реализован в системе System R*. Этот эталонный алгоритм мы и опишем ниже.

ЛГО в каждом узле строится и модифицируется следующим образом:

1) поскольку каждый узел получает потенциальные циклы взаимоблокировки от других узлов, эти ребра добавляются в ЛГО;
2) ребра ЛГО, показывающие, что локальные транзакции ждут завершения транзакций в других узлах, объединяются с ребрами ЛГО, обозначающими удаленные транзакции, которые ожидают завершения локальных.

Пример 5.3. Рассмотрим пример на рис. 5.5. Локальные ГО в обоих узлах модифицируются, как показано на рис. 5.7. ♦

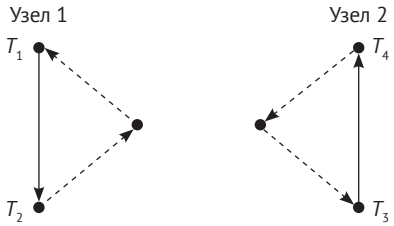


Рис. 5.7 ❖ Модифицированные ЛГО

Локальные детекторы взаимоблокировок ищут две вещи. Если существует цикл, не включающий внешних ребер, значит, имеется локальная взаимобло-

кировка, которую можно локально же и обработать. В противном случае имеется потенциальная распределенная взаимоблокировка, и информацию об этом цикле следует передать другим детекторам. В примере 5.3 возможность такой распределенной взаимоблокировки будет обнаружена обоими узлами.

Теперь возникает вопрос, кому передавать информацию. Очевидно, ее можно передать всем детекторам взаимоблокировок в системе. В отсутствие дополнительной информации это единственный вариант, но ему сопутствуют высокие накладные расходы. Однако если известно, какая транзакция в цикле предшествующая, а какая следующая, то информацию можно посылать вперед или назад по ходу цикла. Тогда принимающий узел модифицирует свой ЛГО, как описано ниже, и проверяет наличие взаимоблокировок. Понятно, что нет нужды передавать информацию в обоих направлениях цикла взаимоблокировки. В примере 5.3 узел 1 передал бы ее узлу 2 как по ходу, так и против хода цикла.

Алгоритмы распределенного обнаружения взаимоблокировок требуют одинаковых модификаций диспетчеров блокировок в каждом узле. Благодаря такой однородности их проще реализовать. Но остается возможность передачи избыточных сообщений. Это происходит, например, в примере 5.3: узел 1 посылает свою информацию о потенциальной взаимоблокировке узлу 2, а тот посылает свою информацию узлу 1. В таком случае взаимоблокировка будет обнаружена обоими детекторами. Помимо передачи лишней информации, возникает дополнительная проблема – каждый узел может выбрать свою жертву для отмены. Алгоритм Обермарка решает эту проблему с помощью временных меток транзакций (монотонно возрастающих счетчиков, детали см. в следующем разделе) и следующего правила. Пусть путь, на котором может возникнуть распределенная взаимоблокировка в локальном ГО узла, имеет вид $T_i \to \ldots \to T_j$. Локальный детектор взаимоблокировок передает информацию о цикле, только если временная метка T_i меньше временной метки T_j. При этом среднее количество передаваемых сообщений уменьшается вдвое. В примере 5.3 путь в узле 1 имеет вид $T_1 \to T_2 \to T_3$, а в узле 2 – $T_3 \to T_4 \to T_1$. В предположении, что индексы транзакций совпадают с их временными метками, только узел 1 передаст информацию узлу 2.

5.2.2. Алгоритмы на основе временных меток

Алгоритмы управления конкурентностью на основе временных меток заранее выбирают порядок сериализации и соответственно выполняют транзакции. Чтобы выбрать такой порядок, диспетчер транзакций назначает каждой транзакции T_i уникальную *временную метку*, $ts(T_i)$, в момент ее инициализации.

Назначение временных меток в распределенной СУБД требует внимания, потому что этим занимаются несколько узлов, так что обеспечить уникальность и монотонность во всей системе нелегко. Один из способов – использовать глобальный (в пределах системы) монотонно возрастающий счетчик. Однако поддержание глобальных счетчиков в распределенной системе – нетривиальная задача. Поэтому предпочтительнее, чтобы каждый узел авто-

номно назначал временные метки, исходя из своего локального счетчика. Для обеспечения уникальности каждый узел дописывает собственный идентификатор в конец счетчика. Поэтому временная метка является кортежем вида ⟨значение локального счетчика, идентификатор узла⟩. Заметим, что идентификатор узла занимает младшую позицию и, следовательно, служит только для упорядочения временных меток транзакций, которым присвоены одинаковые значения локального счетчика. Если каждая система имеет доступ к собственным системным часам, то можно использовать показания часов вместо счетчиков.

Архитектурно (см. рис. 5.2) диспетчер транзакций отвечает за назначение временной метки каждой транзакции и присоединение этой метки к каждой операции базы данных, передаваемой планировщику. Последний отвечает за отслеживание временных меток чтения и записи и проверку сериализуемости.

5.2.2.1. Базовый алгоритм упорядочения временных меток

В базовом алгоритме упорядочения временных меток (УВМ, англ. TO – timestamp ordering) координирующий ДТ назначает временную метку каждой транзакции T_i [$ts(T_i)$], определяет, в каких узлах хранятся элементы данных, и передает релевантные операции этим узлам. Это прямолинейная реализация правила УТ.

Правило УВМ. Пусть даны две конфликтующие операции O_{ij} и O_{kl}, принадлежащие соответственно транзакциям T_i и T_k. Тогда O_{ij} выполняется раньше O_{kl} тогда и только тогда, когда $ts(T_i) < ts(T_k)$. В этом случае говорят, что T_i – *старшая* транзакция, а T_k – *младшая*.

Планировщик, поддерживающий правило УВМ, проверяет каждую новую транзакцию на наличие операций, конфликтующих с уже запланированными. Если новая операция принадлежит транзакции, которая младше, чем все уже запланированные конфликтующие с ней, то операция одобряется, в противном случае отвергается, и транзакция перезапускается с *новой* временной меткой.

Чтобы упростить проверку правила УВМ, каждому элементу данных назначается две временные метки: *временная метка чтения* [$rts(x)$] – наибольшая из временных меток транзакций, которые читали x, и *временная метка записи* [$wts(x)$] – наибольшая из временных меток транзакций, которые записывали (обновляли) x. Теперь достаточно сравнить временную метку операции с временными метками чтения и записи элемента данных, к которому операция хочет получить доступ, и определить, существует ли транзакция с большей временной меткой, которая уже получила доступ к этому элементу.

Базовый алгоритм УВМ-диспетчера транзакций (ВТО-ТМ) представлен в алгоритме 5.4. Истории в каждом узле просто обеспечивают соблюдение правила УВМ. Алгоритм планировщика приведен в алгоритме 5.5. Диспетчер данных тот же, что в алгоритме 5.3. Используются те же структуры данных и предположения, что в централизованных алгоритмах 2PL.

Алгоритм 5.4. Базовое упорядочение временных меток (BTO-TM)

```
Вход: msg : сообщение
begin
    repeat
        ждать msg
        switch тип msg do
            case операция транзакции do          {операция из прикладной программы}
                обозначим операцию op
                switch op.Type do
                    case BT do
                        S ← ∅;                   {S: множество узлов, в которых выполняется
                                                  транзакция}
                        назначить транзакции временную метку – назовем ее ts(T)
                        ПД(op)    {вызвать процессор данных, передав операцию}
                    end case
                    case R, W do
                        определить узел, в котором хранится запрошенный
                        элемент данных (скажем, S_i)
                        BTO-SC_Si (op, ts(T));          {отправить op и ts ПЛ в S_i}
                        S ← S ∪ S                       {построить список узлов, в которых
                                                        выполняется транзакция}
                    end case
                    case A, C do     {отправить op ПД, выполняющим транзакцию}
                        DP_S(op)
                    end case
                end switch
            end case
            case ответ ПЛ do                        {операция была отвергнута ПЛ}
                op.Type ← A;                        {подготовить сообщение об отмене}
                BTO-SCS(op,–) ;                     {спросить у другого участвующего ПЛ}
                перезапустить транзакцию с новой временной меткой
            end case
            case ответ DP do                        {сообщение о завершении операции}
                switch тип операции транзакции do
                    обозначим операцию op
                    case R do вернуть op.val приложению
                    case W do проинформировать приложение о завершении записи
                    case C do
                        if от всех участников получено сообщение о фиксации then
                            проинформировать приложение об успешном
                            завершении транзакции
                        else                        {ждать, когда придут сообщения о фиксации
                                                    от всех участников}
                            протоколировать приход сообщения о фиксации
                        end if
                    end case
                    case A do
                            проинформировать приложение о завершении отмены
                            BTO-SC(op)              {необходимо переустановить ts
                                                    чтения и записи}
                    end case
                end switch
            end case
        end switch
    until бесконечно
end
```

Алгоритм 5.5. Базовый планировщик с упорядочением временных меток (BTO-SC)

```
Вход: op : Op; ts(T) : T временная метка
begin
    извлечь rts(op.arg) и wts(op.arg)
    сохранить rts(op.arg) и wts(op.arg);        {могут понадобиться в случае отмены}
    switch op.arg do
        case R do
            if ts(T) > wts(op.arg) then
                ПД(op);              {операцию можно выполнить; отправить ее ПД}
                rts(op.arg) ← ts(T)
            else
                отправить сообщение «Отвергнуть транзакцию»
                координирующему ДТ
            end if
        end case
        case W do
            if ts(T) > rts(op.arg) и ts(T) > wts(op.arg) then
                ПД(op);              {операцию можно выполнить; отправить ее ПД}
                rts(op.arg) ← ts(T)
                wts(op.arg) ← ts(T)
            else
                отправить сообщение «Отвергнуть транзакцию»
                координирующему ДТ
            end if
        end case
        case A do
            forall op.arg, к которым транзакция получила доступ do
                сбросить rts(op.arg) и wts(op.arg) в начальные значения
            end forall
        end case
    end switch
end
```

Когда планировщик отвергает операцию, диспетчер транзакций перезапускает соответствующую транзакцию с новой временной меткой. Тем самым гарантируется, что у транзакции будет шанс выполниться при следующей попытке. Поскольку транзакции никогда не ждут, удерживая права доступа к элементам данных, базовый алгоритм УВМ никогда не приводит к взаимоблокировкам. Однако за свободу от взаимоблокировок приходится расплачиваться возможностью многократного перезапуска транзакций. В следующем разделе мы обсудим альтернативу базовому алгоритму УВМ, которая уменьшает количество перезапусков.

Еще одна деталь, которую не следует упускать из виду, – взаимодействие между планировщиком и процессором данных. Когда одобренная операция передается процессору данных, планировщик должен воздержаться от передачи другой конфликтующей, но в остальном приемлемой операции процессору данных, пока первая не будет обработана и подтверждена. Это

требование призвано гарантировать, что процессор данных выполняет операции в том порядке, в каком их передает планировщик. В противном случае значения временных меток чтения и записи в элементе данных, к которому получен доступ, были бы неверны.

Пример 5.4. Предположим, что УВМ-планировщик сначала получает $W_i(x)$, а затем $W_j(x)$, где $ts(T_i) < ts(T_j)$. Планировщик одобрит обе операции и передаст их процессору данных. После этих двух операций окажется, что $wts(x) = ts(T_j)$, и мы ожидаем, что в базе данных будет представлен результат $W_j(x)$. Однако если процессор данных выполнит их не в таком порядке, то результат в базе данных будет неверным. ♦

Планировщик может обеспечить нужный порядок, организовав очередь для каждого элемента данных, цель которой – задержать передачу одобренной операции до тех пор, пока не будет получено подтверждение от процессора данных, относящееся к предыдущей операции над тем же элементом данных. Эта деталь в алгоритме 5.5 не показана.

Такое осложнение не возникает в алгоритмах на основе 2PL, потому что диспетчер блокировок сам упорядочивает операции, освобождая блокировки только после того, как операция выполнена. В некотором смысле очередь, организованную УВМ-планировщиком, можно рассматривать как блокировку. Однако это не означает, что истории, порожденные УВМ-планировщиком и 2PL-планировщиком, всегда эквивалентны. Существуют истории, которые УВМ-планировщик мог бы породить, а 2PL-планировщик – нет.

Напомним, что в случае строгих алгоритмов на основе 2PL освобождение блокировок дополнительно откладывается до фиксации или отмены транзакции. Можно разработать строгий УВМ-алгоритм, используя подобную схему. Например, если $W_i(x)$ одобрена и передана процессору данных, то планировщик откладывает все операции $R_j(x)$ и $W_j(x)$ (для всех T_j) до завершения T_i (фиксации или отмены).

5.2.2.2. Консервативный УВМ-алгоритм

В предыдущем разделе мы отметили, что базовый УВМ-алгоритм никогда не заставляет операции ждать, а вместо этого перезапускает их. Мы также подчеркнули, что хотя это можно считать преимуществом в силу отсутствия взаимоблокировок, это также и недостаток, поскольку многочисленные перезапуски плохо сказываются на производительности. Консервативные УВМ-алгоритмы пытаются снизить эти издержки, уменьшив количество перезапусков.

Сначала опишем метод, который обычно используется, чтобы уменьшить вероятность перезапуска. Напомним, что УВМ-планировщик перезапускает транзакцию, если уже запланирована или была выполнена более молодая конфликтующая транзакция. Заметим, что количество таких случаев резко возрастает, если, например, один узел малоактивен по сравнению с другими и на протяжении длительного периода не запускает никаких транзакций. В таком случае его счетчик, выступающий в роли временной метки, оказывается намного меньше счетчиков других узлов. Если ДТ в этом узле затем

получает транзакцию, то операции, отправленные в истории на других узлах, почти наверняка будут отвергнуты, что приведет к перезапуску транзакции. Более того, эта транзакция будет перезапускаться раз за разом, пока счетчик в инициировавшем ее узле не сравняется со счетчиками других узлов.

Описанный сценарий показывает, что полезно синхронизировать счетчики во всех узлах. Однако полная синхронизация не только обходится дорого – поскольку требует обмена сообщениями при каждом изменении счетчика, – но и не обязательна. Вместо этого каждый диспетчер транзакций может посылать свои удаленные операции, а не истории, диспетчерам транзакций в других узлах. Тогда принимающие диспетчеры транзакций смогут сравнить свои счетчики со счетчиком в поступившей операции. Если значение локального счетчика меньше, то диспетчер корректирует свой счетчик, делая его на единицу больше поступившего. Тем самым гарантируется, что ни один счетчик в системе не сможет значительно опередить или отстать от остальных. Конечно, если вместо счетчиков используются системные часы, то такая приближенная синхронизация достигается автоматически, при условии что сами часы синхронизируются по протоколу типа Network Time Protocol (NTP).

Консервативные УВМ-алгоритмы выполняют операции иначе, чем базовый. Базовый УВМ-алгоритм пытается выполнить операцию, сразу как только она одобрена; поэтому его можно назвать «агрессивным» или «прогрессивным». Консервативные же алгоритмы задерживают каждую операцию до тех пор, пока не будет уверенности, что планировщику не поступит операция с меньшей временной меткой. Если выполнение этого условия нельзя гарантировать, то планировщик никогда не отвергнет операцию. Однако такая задержка открывает возможность для взаимоблокировок.

Базовая техника, применяемая в консервативных УВМ-алгоритмах, основана на следующей идее: операции каждой транзакции буферизуются, пока не будет установлен порядок, при котором операция не может быть отвергнута, и затем выполняются в этом порядке. Мы рассмотрим одну из возможных реализаций консервативного УВМ-алгоритма.

Предположим, что каждый планировщик поддерживает по одной очереди для каждого имеющегося в системе диспетчера транзакций. Планировщик в узле s хранит все операции, получаемые от диспетчера транзакций в узле t, в очереди Q_s^t. Планировщик в узле s имеет по одной такой очереди для каждого узла t. Когда приходит операция от диспетчера транзакций, она помещается в соответствующую очередь в порядке возрастания временных меток. Истории в каждом узле выбирают для выполнения операции из этих очередей также в порядке возрастания временных меток.

Эта схема уменьшает количество перезапусков, но не гарантирует их полного исчезновения. Рассмотрим случай, когда в узле s очередь узла t (Q_s^t) пуста. Планировщик в узле s выберет операцию [скажем, $R(x)$] с наименьшей временной меткой и передаст ее процессору данных. Однако узел t, возможно, послал s операцию [скажем, $W(x)$] с меньшей временной меткой, которая просто еще не успела дойти. Когда эта операция достигнет узла s, она будет отвергнута, т. к. нарушается правило УВМ: она хочет обратиться к элементу данных, к которому уже получила доступ (в несовместимом режиме) операция с большей временной меткой.

Можно спроектировать чрезвычайно консервативный УВМ-алгоритм, настаивая на том, чтобы планировщик выбирал операцию для передачи процессору данных, только если в каждой очереди есть хотя бы одна операция. Это гарантирует, что любая операция, которую планировщик получит в будущем, будет иметь временную метку, большую или равную тем, что уже находятся в очередях. Конечно, если у диспетчера транзакций нет транзакций, ожидающих обработки, то он должен периодически отправлять фиктивные сообщения всем планировщикам в системе, уведомляя их о том, что в будущих операциях временные метки будут заведомо больше, чем в фиктивном сообщении.

Внимательный читатель наверняка заметил, что чрезвычайно консервативный УВМ-планировщик на самом деле выполняет транзакции в каждом узле строго последовательно. Это очень сильное ограничение. Для его преодоления было предложено группировать транзакции в классы. Классы транзакций определяются, исходя из их множеств чтения и записи. Поэтому чтобы определить, к какому классу принадлежит транзакция, достаточно сравнить множество чтения и множество записи транзакции с соответствующими множествами каждого класса. Таким образом, консервативный УВМ-алгоритм можно модифицировать, так чтобы в каждом узле присутствовала очередь не для каждого диспетчера транзакций, а для каждого класса транзакций. Альтернативно можно пометить каждую очередь классом, которому она принадлежит. В обоих случаях изменяются условия передачи операции процессору данных. Больше не обязательно ждать появления хотя бы одной операции в каждой очереди; достаточно, если будет хотя бы одна операция в каждом классе, которому принадлежит транзакция. Можно определить и другие более слабые, но достаточные условия, позволяющие сократить время ожидания. Вариант этого метода применяется в системе-прототипе SDD-1.

5.2.3. Многоверсионное управление конкурентностью

Все рассмотренные выше подходы относятся к обновлению на месте: когда значение элемента данных обновляется, старое значение заменяется новым в базе данных. Альтернатива – хранить версии обновляемых элементов данных. Этот класс алгоритмов называется *многоверсионным управлением конкурентностью* (multiversion concurrency control – MVCC). Тогда каждая транзакция «видит» то значение элемента данных, которое соответствует уровню ее изоляции. Наличие нескольких версий базы данных открывает также возможность для *путешествий во времени*, т. е. запросов, отслеживающих изменение данных со временем. Проблема MVCC – наличие нескольких версий элементов данных с разными значениями. Чтобы сэкономить место, старые версии данных должны периодически удаляться. Но это можно делать, только когда распределенная СУБД уверена, что больше не появится транзакция, которой нужен доступ к удаленным версиям.

Первое предложение на эту тему датируется еще 1978 годом, но популярность идея обрела сравнительно недавно и теперь реализована во многих СУБД, включая IBM DB2, Oracle, SQL Server, SAP HANA, BerkeleyDB, PostgreSQL, а также таких системах, как Spanner. В этих системах обеспечивается *изоляция моментальных снимков* (snapshot isolation), которую мы будем обсуждать в разделе 5.3.

В методе MVCC для обеспечения изоляции транзакций обычно используются временные метки, но в некоторых предложениях многоверсионность надстраивается над уровнем управления конкурентностью на основе блокировок. Здесь мы ограничимся реализацией на базе временных меток, гарантирующей сериализуемость. В этой реализации каждая версия элемента данных помечается временной меткой создавшей его транзакции. Идея в том, что каждая операция чтения обращается к той версии элемента данных, которая соответствует ее временной метке, в результате чего количество отмен и перезапусков транзакций уменьшается. При этом гарантируется, что каждая транзакция работает с тем состоянием базы данных, которое она увидела бы, если бы все транзакции выполнялись строго последовательно в порядке возрастания временных меток.

Существование версий прозрачно для пользователей, которые указывают в транзакциях просто элементы данных, а не конкретные версии. Диспетчер транзакций назначает каждой транзакции временную метку, которая используется также для отслеживания временных меток версий данных. Операции обрабатываются следующим образом, гарантирующим сериализуемость истории:

1) $R_i(x)$ транслируется в чтение одной версии x. Для этого ищется версия x (скажем, x_v) такая, что $ts(x_v)$ – наибольшая временная метка, меньшая $ts(T_i)$. Затем $R_i(x_v)$ передается процессору данных для чтения x_v. Этот случай изображен на рис. 5.8а, где показано, что R_i может прочитать версию (x_v), которую прочитала бы, если бы поступила в порядке, определяемом временной меткой;

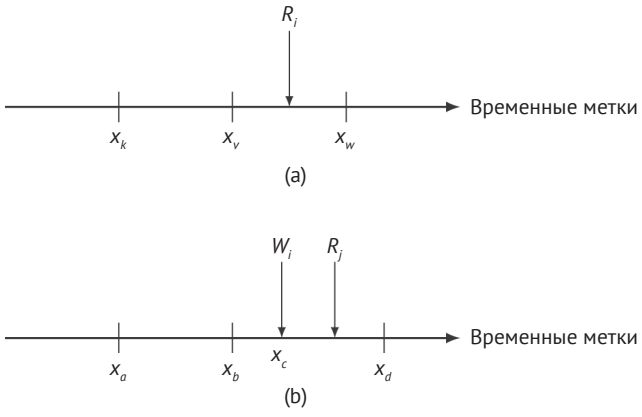


Рис. 5.8 ❖ Случаи многоверсионного УВМ

2) $W_i(x)$ транслируется в $W_i(x_w)$, так что $ts(x_w) = ts(T_i)$, и передается процессору данных тогда и только тогда, когда никакая другая транзакция с временной меткой, большей $ts(T_i)$, не прочитала версию x (скажем, x_r) такую, что $ts(x_r) > ts(x_w)$. Иными словами, если планировщик уже обработал операцию $R_j(x_r)$ такую, что $ts(T_i) < ts(x_r) < ts(T_j)$, то $W_i(x)$ отвергается. Этот случай изображен на рис. 5.8b, где показано, что если бы W_i была одобрена, то она создала бы версию (x_c), которую R_j должна была бы прочесть, но не прочла, потому что этой версии еще не было в момент выполнения R_j, – вместо этого она прочла версию x_b, что приводит к неправильной истории.

5.2.4. Оптимистические алгоритмы

В оптимистических алгоритмах предполагается, что конфликты транзакций и состязание за данные происходят не слишком часто, поэтому транзакциям разрешено выполняться без синхронизации до самого конца, и лишь тогда проверяется корректность. Оптимистические алгоритмы управления конкурентностью могут быть основаны на блокировках или на временных метках (последний вариант содержался в оригинальном предложении). В этом разделе мы опишем распределенный оптимистический алгоритм на основе временных меток.

Каждая транзакция проходит через пять этапов: чтение (R), выполнение (E), запись (W), проверка (V) и фиксация (C). Разумеется, фиксация превращается в отмену, если транзакция не прошла проверку. Этот алгоритм назначает временные метки транзакциям в начале шага проверки, а не в начале транзакции, как в (пессимистических) УВМ-алгоритмах. Кроме того, с элементами данных не ассоциируются временные метки чтения и записи – алгоритм работает только с временными метками транзакций на этапе проверки.

Каждая транзакция T_i разбивается (диспетчером транзакций в инициировавшем узле) на несколько подтранзакций, каждая из которых может выполняться во многих узлах. Будем обозначать T_i^s подтранзакцию T_i, выполняемую в узле s. В начале этапа проверки транзакции назначается временная метка, которая будет также временной меткой всех ее подтранзакций. Локальная проверка T_i^s производится по следующим взаимно исключающим правилам.

Правило 1. В каждом узле s, если все транзакции T_k^s, где $ts(T_k^s) < ts(T_i^s)$, завершили этап записи, до того как T_i^s начала этап чтения (рис. 5.9a), проверка считается успешной, потому что транзакции выполняются последовательно.

Правило 2. В каждом узле s, если существует транзакция T_k^s такая, что $ts(T_k^s) < ts(T_i^s)$, которая завершает свой этап записи, пока T_i^s находится в процессе чтения (рис. 5.9b), проверка считается успешной, если $WS(T_k^s) \cap RS(T_i^s) = \emptyset$.

Правило 3. В каждом узле s, если существует транзакция T_k^s такая, что $ts(T_k^s) < ts(T_i^s)$, которая завершает свой этап чтения, до того как T_i^s завершает свой этап чтения (рис. 5.9c), проверка считается успешной, если $WS(T_k^s) \cap RS(T_i^s) = \emptyset$ и $WS(T_k^s) \cap WS(T_i^s) = \emptyset$.

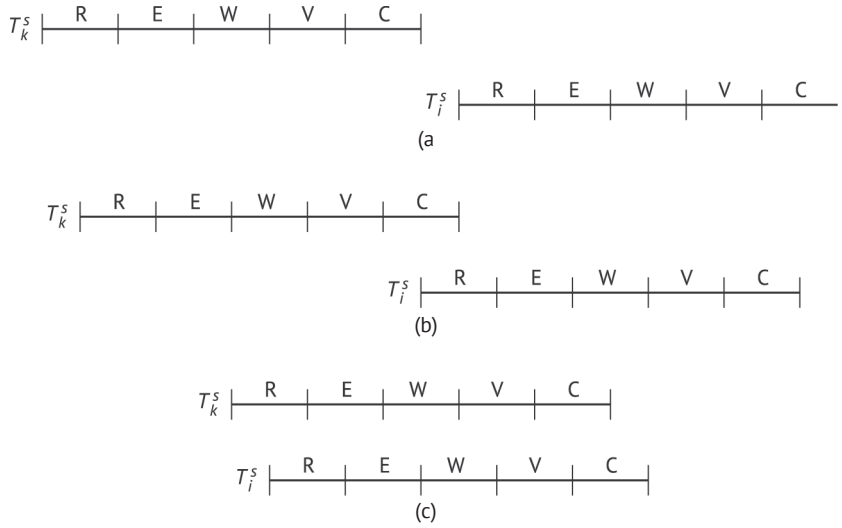


Рис. 5.9 ❖ Возможные сценарии выполнения

Правило 1 очевидно; оно означает, что транзакции фактически выполняются последовательно в порядке своих временных меток. Правило 2 гарантирует, что никакие элементы данных, обновленные T_k^s, не читаются T_i^s и что T_k^s завершает запись своих обновлений в базу данных (т. е. фиксируется), до того как T_i^s начинает запись. Таким образом, обновления T_i^s не будут перезаписаны обновлениями T_k^s. Правило 3 аналогично правилу 2, но не требует, чтобы T_k^s завершила запись, до того как T_i^s начнет запись. Требуется только, чтобы обновления T_k^s не влияли на этап чтения или записи T_i^s.

После того как транзакция проверена локально и оказалось, что она не нарушает согласованность локальной базы данных, необходимо выполнить глобальную проверку и убедиться в соблюдении правила взаимной согласованности. Для этого нужно проверить, что приведенные выше правила соблюдаются в каждом участвующем узле.

Плюс оптимистических алгоритмов в том, что потенциально они допускают более высокий уровень конкурентности. Показано, что если конфликты транзакций очень редки, то оптимистический механизм производительнее блокировки. Трудность же оптимистических подходов связана с поддержанием информации, необходимой для контроля. Для проверки подтранзакции T_i^s необходимо хранить множества чтения и записи завершенных транзакций, которые выполнялись, когда T_i^s поступила в узел s.

Другая проблема – зависание. Предположим, что этап проверки длинной транзакции завершился неудачно. Не исключено, что и последующие попытки проверки не дадут результата. Конечно, проблему можно решить, предоставив транзакции монопольный доступ к базе данных после некоторого количества попыток. Но при этом уровень конкурентности снизится до единственной транзакции. Точная смесь транзакций, которая способна привести к недопустимому уровню перезапусков, – вопрос, который еще предстоит изучить.

5.3. Распределенное управление конкурентностью с помощью изоляции моментальных снимков

До сих пор мы обсуждали алгоритмы, обеспечивающие сериализуемость. Хотя сериализуемость – лучше всего изученный критерий корректности для выполнения конкурентных транзакций, в некоторых приложениях он может считаться слишком строгим, поскольку запрещает истории, которые можно было бы разрешить. В частности, сериализуемость создает узкое место, из-за чего распределенные базы данных не допускают высокой степени масштабирования. Основная причина в том, что слишком сильно ограничивается уровень конкурентности транзакций, т. к. большие запросы чтения конфликтуют с обновлениями. Это привело к определению альтернативы – *изоляции моментального снимка* (snapshot isolation – SI). SI широко применяется в коммерческих системах, на эту технологию опираются многие современные системы, в частности Google Spanner и LeanXcale, которые масштабируются в очень широких пределах; мы обсудим эти подходы в разделе 5.5. Изоляция моментального снимка обеспечивает повторяемое чтение, но не сериализуемую изоляцию. Каждая транзакция «видит» согласованный снимок базы данных, когда начинает работать, и ее операции чтения и записи выполняются на этом снимке. Таким образом, произведенные ей обновления не видны другим транзакциям, и она не видит обновлений, выполненных другими транзакциями с момента начала ее работы.

Изоляция снимков – многоверсионный подход, в котором транзакциям позволено читать подходящий снимок (т. е. версию). Важное достоинство основанного на SI управления конкурентностью заключается в том, что транзакции чтения могут работать без заметных накладных расходов на синхронизацию. Для транзакций обновления алгоритм управления конкурентностью (в централизованной системе) выглядит так:

S1. Когда транзакция T_i начинается, она получает *временную метку начала* $ts_b(T_i)$.

S2. Когда транзакция T_i готова к фиксации, она получает *временную метку фиксации* $ts_c(T_i)$, которая больше любой из уже существующих меток ts_b и ts_c.

S3. T_i фиксирует свои обновления, если не существует никакой другой транзакции T_j такой, что $ts_c(T_j) \in [ts_b(T_i), ts_c(T_i)]$ (т. е. никакая другая транзакция не была зафиксирована с момента начала T_i), в противном случае T_i отменяется. Это правило, называемое *преимуществом первого зафиксировавшего*, предотвращает потерю обновлений.

S4. Когда T_i зафиксирована, выполненные ей изменения становятся доступны всем транзакциям T_k таким, что $ts_b(T_k) > ts_c(T_i)$.

Когда SI используется в качестве критерия корректности при распределенном управлении конкурентностью, возникает проблема, как вычислить

согласованный моментальный снимок (версию), с которым работает транзакция T_i. Если бы множества чтения и записи транзакции были известны заранее, то можно было бы централизованно вычислить снимок (координирующим ДТ), собрав информацию с участвующих узлов. Но это, конечно, нереально. Нам нужна глобальная гарантия, похожая на гарантию сериализуемости, рассмотренную выше. Иными словами:

1) каждая локальная история обладает свойством SI и
2) глобальная история обладает свойством SI, т. е. порядок фиксации транзакций во всех узлах одинаков.

Теперь мы выясним, какие условия должны выполняться, чтобы можно было реализовать только что описанную гарантию. Начнем с определения *отношения зависимости* между двумя транзакциями. Оно важно в этом контексте, потому что снимок, который читает транзакция T_i, должен включать только обновления транзакций, от которых она зависит. Транзакция T_i в узле s (T_i^s) зависит от T_j^s, и это обозначается *dependent*(T_i^s, T_j^s) тогда и только тогда, когда ($RS(T_i^s) \cap WS(T_j^s) \neq \emptyset$) $\vee$ ($WS(T_i^s) \cap RS(T_j^s) \neq \emptyset$) $\vee$ ($WS(T_i^s) \cap WS(T_j^s) \neq \emptyset$). Если такая зависимость имеет место в любом участвующем узле, то имеет место зависимость *dependent*(T_i, T_j).

Теперь мы можем более точно описать условия, которые должны удовлетворяться, чтобы гарантировать глобальную изоляцию снимков в определенном выше смысле. Условия ниже сформулированы для пар транзакций, но в силу транзитивности они имеют место для множества транзакций. Чтобы транзакция T_i видела глобально согласованный моментальный снимок, для каждой пары транзакций должны выполняться следующие условия:

C1. Если *dependent* (T_i, T_j) $\wedge$ $ts_b(T_i^s) < ts_c(T_j^s)$, то $ts_b(T_i^t) < ts_c(T_j^t)$ в каждом узле t, где T_i и T_j выполняются вместе.

C2. Если *dependent* (T_i, T_j) $\wedge$ $ts_c(T_i^s) < ts_b(T_j^s)$, то $ts_c(T_i^t) < ts_b(T_j^t)$ в каждом узле t, где T_i и T_j выполняются вместе.

C3. Если $ts_c(T_i^s) < ts_c(T_j^s)$, то $ts_c(T_{ti}) < ts_b(T_j^t)$ в каждом узле t, где T_i и T_j выполняются вместе.

Первые два условия гарантируют, что *dependent*(T_i, T_j) истинно во всех узлах, т. е. T_i всегда корректно видит это отношение зависимости во всех узлах. Третье условие гарантирует, что порядок фиксации транзакций одинаков во всех участвующих узлах, и предотвращает включение в два снимка частичных фиксаций, несовместимых между собой.

Прежде чем переходить к обсуждению SI-алгоритма управления конкурентностью, определим, какая информация хранится в каждом узле s.

- Для любой активной транзакции T_i множество *активных* и *зафиксированных* транзакций в s разбивается на две группы: конкурентные с T_i (т. е. все такие T_j, что $ts_b(T_i^s) < ts_c(T_j^s)$), и последовательные (т. е. все такие T_j, что $ts_c(T_j^s) < ts_b(T_i^s)$); заметим, что последовательная – не то же самое, что зависимая, локальная история обозначает порядок в локальной истории в узле s, не делая никаких утверждений о зависимости.
- Монотонно возрастающие отсчеты времени событий.

Базовый распределенный SI-алгоритм по существу реализует шаг S3 централизованного алгоритма, представленного выше (хотя существуют и другие

реализации), т. е. решает, следует ли зафиксировать или отменить транзакцию T_i. Алгоритм работает следующим образом.

D1. Координирующий ДТ транзакции T_i просит каждый участвующий узел s прислать свое множество транзакций, конкурентных с T_i. Он включает в это сообщение собственный отсчет событий.

D2. Каждый узел s отвечает на запрос координирующего ДТ, посылая свое локальное множество транзакций, конкурентных с T_i.

D3. Координирующий ДТ объединяет все локальные множества конкурентных транзакций в одно глобальное множество конкурентных транзакций для T_i.

D4. Координирующий ДТ рассылает этот глобальный список конкурентных транзакций всем участвующим узлам.

D5. Каждый узел s проверяет выполнение условий C1 и C2. Для этого он смотрит, существует ли в глобальном списке конкурентных транзакций транзакция T_j, присутствующая в последовательном списке локальной истории (т. е. в локальной истории s T_j выполнилась раньше T_i), и такая, что T_i от нее зависит (т. е. имеет место $dependent(T_i^s, T_j^s)$). Если да, то T_i не видит согласованного снимка в узле s, т. е. должна быть отменена. В противном случае T_i успешно прошла проверку в узле s.

D6. Каждый узел s отправляет положительный или отрицательный результат проверки координирующему ДТ. Отправляя положительный результат, узел s обновляет свой отсчет событий, присваивая ему максимум из своего отсчета и полученного от ДТ отсчета; этот новый отсчет включается в ответное сообщение.

D7. Если координирующий ДТ получает хотя бы один отрицательный результат проверки, то T_i отменяется, потому что существует по меньшей мере один узел, который не видит согласованного списка. В противном случае координирующий ДТ глобально одобряет T_i и разрешает ей фиксировать свои изменения. Если принято решение о глобальном одобрении, то координирующий ДТ обновляет свой отсчет событий, присваивая ему максимум из всех отсчетов, полученных от участвующих узлов, и своего собственного отсчета.

D8. Координирующий ДТ уведомляет все участвующие узлы о том, что T_i успешно проверена и может быть зафиксирована. Он также присоединяет к сообщению новое значение своего отсчета событий, которое становится меткой $ts_c(T_i)$.

D9. Получив это сообщение, каждый участвующий узел s делает обновления T_i постоянными и обновляет свой отсчет событий, как и раньше.

В этом алгоритме проверка условий C1 и C2 производится на шаге D5; остальные шаги служат для сбора необходимой информации и координации одобрения. Синхронизация отсчетов событий между узлами нужна для обеспечения условия C3, поскольку гарантирует, что порядок фиксации зависимых транзакций согласован между узлами, так что и глобальный снимок будет согласован.

Рассмотренный нами алгоритм – один из возможных подходов к реализации SI в распределенной СУБД. Он гарантирует глобальную изоляцию

снимка, но требует, чтобы глобальный снимок был вычислен заранее, а значит, создает узкое место в плане масштабируемости. Например, отправка всех конкурентных транзакций на шаге D2, очевидно, не масштабируется, потому что в системе, выполняющей миллионы конкурентных транзакций, пришлось бы отправлять миллионы транзакций при каждой проверке, что резко ограничило бы масштабируемость. Кроме того, требуется, чтобы все транзакции проходили один и тот же процесс одобрения. Это можно оптимизировать, отделив транзакции, обращающиеся к данным только в одном узле, которые, следовательно, не требуют порождения глобального снимка, от глобальных транзакций, выполняемых в нескольких узлах. Один из способов сделать это – инкрементно строить снимок, который читает транзакция T_i по мере того, как обращается к данным в разных узлах.

5.4. Надежность распределенных СУБД

В централизованных СУБД возможны ошибки трех типов: ошибки транзакций (например, отмена транзакции), отказы узлов (это приводит к потере данных в памяти, но не в постоянном хранилище) и сбои носителей (это может привести к частичной или полной потере данных в постоянном хранилище). В распределенной среде система должна иметь дело с четвертым типом ошибок: *ошибками передачи данных*. Есть несколько видов таких ошибок; наиболее распространенными являются ошибки в сообщениях, неправильно упорядоченные сообщения, потерянные (или не могущие быть доставленными) сообщения и ошибки линий связи. В ошибках первых двух типов повинна компьютерная сеть, на них мы останавливаться не будем. Поэтому при обсуждении надежности распределенной СУБД мы ожидаем от сети гарантий, что два сообщения, отправленных процессом в одном узле процессу в другом узле, будут доставлены без ошибок и в том порядке, в каком были отправлены, т. е. считается, что канал связи – надежная FIFO-очередь.

Потеря или невозможность доставки сообщений обычно является следствием сбоев в линиях связи или отказа узла-приемника. Если имеет место сбой линии связи, то, помимо потери некоторых сообщений, находившихся в этот момент в канале, сеть может оказаться разбитой на две или более несообщающихся групп. Это называется *разделением сети*. Если сеть разделена, то узлы, находящиеся в каждом разделе, могут продолжать работу. В таком случае проблемой становится доступ к данным, хранящимся в нескольких разделах. В распределенной системе обычно невозможно различить ошибки из-за отказа узла-приемника и из-за сбоя линии связи. В обоих случаях узел-источник отправляет сообщение, но не получает ответа в течение ожидаемого времени, это называется *тайм-аутом*. В этот момент должны вступить в игру алгоритмы обеспечения надежности.

Ошибки передачи данных высвечивают уникальный аспект ошибок в распределенных системах. Состояние централизованной системы можно охарактеризовать словами «все или ничего»: система либо работает, либо нет. Таким образом, ошибки фатальны: если случается отказ, то вся система ока-

зывается неработоспособна. Очевидно, что в распределенной системе это не так. Как мы уже неоднократно отмечали, в этом их потенциальная сила. Однако это также усложняет проектирование алгоритмов управления транзакциями, поскольку если сообщение не доставлено, то трудно понять, то ли отказал узел-приемник, то ли сообщение не пришло из-за сбоя в сети.

Если сообщения невозможно доставить, то мы предполагаем, что сеть ничего не может с этим поделать. Она не буферизует их для доставки получателю, когда обслуживание восстановится, и не уведомляет процесс-отправитель о том, что сообщение не может быть доставлено. Короче говоря, сообщение просто теряется. Такое предположение отражает минимальные требования к сети и возлагает ответственность за обработку ошибок на распределенную СУБД. Следовательно, распределенная СУБД должна обнаруживать невозможность доставки сообщения. Механизм обнаружения обычно зависит от характеристик коммуникационной системы, в которой реализована распределенная СУБД. Детали выходят за рамки этой книги, мы, как уже было сказано, будем предполагать, что отправитель сообщения взводит таймер и ждет до истечения периода тайм-аута, после чего решает, что сообщение не было доставлено.

Протоколы обеспечения надежности в распределенной системе должны обеспечить атомарность и долговечность распределенных транзакций, выполняемых в нескольких базах данных. Эти протоколы касаются распределенного выполнения команд `Begin_transaction`, `Read`, `Write`, `Abort`, `Commit` и `recover`. Команды `Begin_transaction`, `Read` и `Write` выполняются диспетчерами локального восстановления (ДЛВ) так же, как в централизованных СУБД. Особого внимания в распределенных СУБД требуют команды `Commit`, `Abort` и `Read`. Фундаментальная трудность заключается в том, чтобы гарантировать, что все узлы, участвующие в выполнении транзакции, приходят к одному и тому же решению о судьбе транзакции (зафиксировать или отменить).

Реализация протокола обеспечения надежности в распределенной системе в рамках принятой в этой книге архитектурной модели поднимает ряд интересных и трудных вопросов. Мы обсудим их в разделе 5.4.6 после введения в эти протоколы. А пока примем типичную абстракцию: будем предполагать, что в инициировавшем транзакцию узле существует процесс-*координатор*, а в каждом узле, где транзакция выполняется, работают процессы-*участники*. Таким образом, протоколы обеспечения надежности в распределенной системе реализованы между координатором и участниками.

Механизмы обеспечения надежности в распределенных системах баз данных состоят из протоколов фиксации, завершения и восстановления – протоколы фиксации и восстановления описывают, как выполняются команды `Commit` и `recover`, а протоколы завершения – как продолжающие работать узлы могут завершить транзакцию, обнаружив, что какой-то узел вышел из строя. Протоколы завершения и восстановления – две стороны проблемы восстановления: при наличии отказа узла протоколы завершения описывают, как должны обработать ошибки оставшиеся в живых узлы, а протоколы восстановления описывают процедуры, которые должен выполнить процесс (координатор или участник) в отказавшем узле, чтобы восстановить его со-

стояние после перезапуска. В случае разделения сети протоколы завершения принимают необходимые меры, чтобы завершить активные транзакции, выполняемые в разных разделах, а протоколы восстановления решают проблему, как привести базу данных в глобально согласованное состояние, после возобновления связности сети.

Основное требование протоколов фиксации – обеспечить атомарность распределенных транзакций. Это означает, что, несмотря на то что распределенная транзакция охватывает несколько узлов и некоторые из них могут отказать во время выполнения, все равно результатом транзакции в распределенной базе является все или ничего. Это называется *атомарной фиксацией*. Мы предпочли бы, чтобы протоколы завершения были *неблокирующими*. Протокол является неблокирующим, если он позволяет транзакции завершиться в работающих узлах, не дожидаясь восстановления отказавшего. Мы также хотели бы, чтобы протоколы распределенного восстановления были *независимыми*. Независимые протоколы восстановления определяют, как завершить транзакцию, которая выполнялась в момент отказа, не консультируясь ни с какими другими узлами. Существование таких протоколов сократило бы количество сообщений, которыми узлы обмениваются в процессе восстановления. Заметим, что из существования независимых протоколов восстановления следует существование неблокирующих протоколов завершения, но обратное неверно.

5.4.1. Протокол двухфазной фиксации

Протокол двухфазной фиксации (2PC) – очень простой и элегантный протокол, который гарантирует атомарную фиксацию распределенных транзакций. Он обобщает последствия локальной атомарной фиксации на распределенные транзакции, настаивая, чтобы все узлы, участвующие в выполнении распределенной транзакции, согласились зафиксировать транзакцию, прежде чем ее результаты станут постоянными. Существует несколько причин, по которым такая синхронизация узлов необходима. Во-первых, в зависимости от используемого алгоритма управления конкурентностью некоторые планировщики могут быть не готовы завершить транзакцию. Например, если транзакция прочитала значение элемента данных, обновленного другой, еще не зафиксированной транзакцией, то соответствующий планировщик может не захотеть фиксировать первую транзакцию. Конечно, строгие алгоритмы управления конкурентностью, которые избегают каскадной отмены, не позволили бы прочитать обновленное значение никакой другой транзакции, пока та, что произвела обновление, не завершится. Иногда это называется *условием возможности восстановления*.

Еще одна причина, по которой участник может не согласиться на фиксацию, – взаимоблокировки, которые вынуждают участника отменить транзакцию. Заметим, что в этом случае участнику должно быть разрешено отменить транзакцию без указания со стороны координатора. Эта возможность очень важна и называется *односторонней отменой*. Другая причина односторонней отмены – тайм-ауты, о ней мы поговорим ниже.

Ниже приведено краткое описание протокола 2PC без учета ошибок. В обсуждении нам поможет диаграмма переходов состояний на рис. 5.10. Состояния обозначены окружностями, а переходы состояний – соединяющими их ребрами. Завершающие состояния обозначены концентрическими окружностями. Метки на ребрах интерпретируются следующим образом: сверху указана причина перехода состояний, т. е. полученное сообщение, а снизу – сообщение, отправленное в результате перехода.

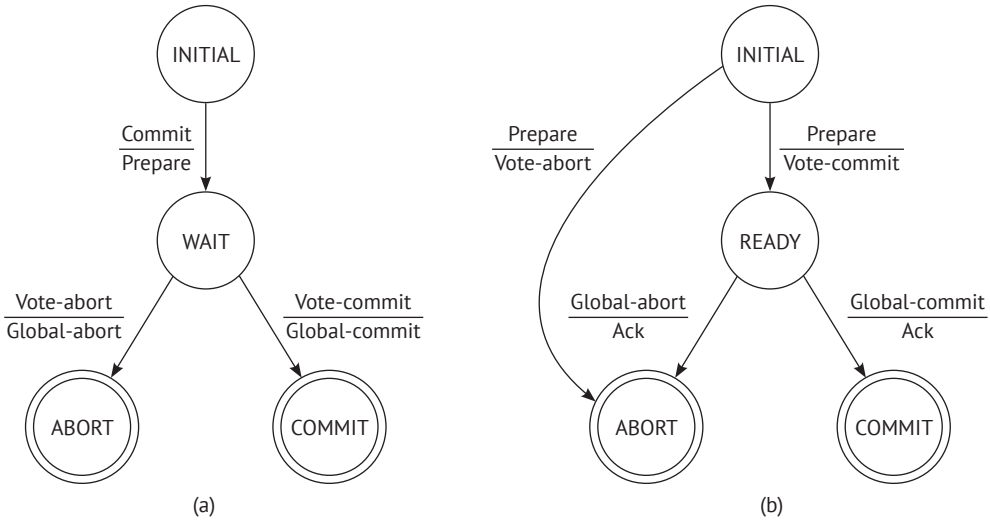


Рис. 5.10 ❖ Переходы состояний в протоколе 2PC:
(a) состояния координатора; (b) состояния участников

1. В самом начале координатор помещает запись `begin_commit` в свой журнал, отправляет сообщение «prepare» (подготовиться) всем участникам и переходит в состояние WAIT (ожидание).
2. Получив сообщение «prepare», участник проверяет, может ли он зафиксировать транзакцию. Если да, то участник помещает запись `ready` в журнал, отправляет сообщение «vote-commit» (голосую за фиксацию) координатору и переходит в состояние READY (готовность); в противном случае участник помещает в журнал запись `abort` и отправляет координатору сообщение «vote-abort» (голосую за отмену).
3. Если узел принял решение об отмене, то он может забыть об этой транзакции, поскольку такое решение означает вето (одностороннюю отмену).
4. Получив ответы от всех участников, координатор решает, зафиксировать или отменить транзакцию. Если хотя бы один участник проголосовал за отмену, то координатор обязан отменить транзакцию глобально. Поэтому он помещает в журнал запись `abort`, отправляет всем участникам сообщение «global-abort» (глобальная отмена) и переходит в состояние ABORT. В противном случае он помещает в журнал запись `commit`, отправляет всем участникам сообщение «global-commit» и переходит в состояние COMMIT.

5. Участники фиксируют или отменяют транзакцию, следуя инструкциям координатора, и отправляют ему подтверждение. В этот момент координатор завершает транзакцию, помещая запись `end_of_transaction` в журнал.

Обратите внимание, как именно координатор приходит к решению о глобальном завершении транзакции. Это решение регулируется двумя правилами, совокупность которых называют *правилом глобальной фиксации*.

1. Если хотя бы один участник голосует за отмену транзакции, то координатор обязан принять решение о глобальной отмене.
2. Если все участники голосуют за фиксацию транзакции, то координатор обязан принять решение о глобальной фиксации.

Взаимодействие координатора и одного участника по протоколу 2PC в отсутствие ошибок показано на рис. 5.11, где окружности обозначают состояния, а штриховые линии – сообщения между координатором и участниками. Метки рядом со штриховыми линиями описывают природу сообщений.

На основании рис. 5.11 можно сделать несколько важных замечаний о протоколе 2PC. Во-первых, 2PC разрешает участнику отменить транзакцию в одностороннем порядке до того, как он проголосовал утвердительно. Во-вторых, если участник проголосовал за фиксацию или отмену, то изменить свое решение он не может. В-третьих, находясь в состоянии READY, участник может либо зафиксировать, либо отменить транзакцию в зависимости от того, какое сообщение придет от координатора. В-четвертых, решение о глобальном завершении принимается координатором по правилу глобальной фиксации. Наконец, отметим, что в некоторых состояниях процессы координатора и участника ждут сообщений друг от друга. Чтобы гарантировать выход из этих состояний и завершение, используются таймеры. Каждый процесс при входе в состояние взводит таймер, и если ожидаемое сообщение не пришло до его срабатывания, то процесс вызывает свой протокол тайм-аута (который мы обсудим ниже).

Протокол 2PC можно реализовать несколькими способами. Выше мы обсуждали и продемонстрировали на рис. 5.11 *централизованный 2PC*, который называется так, потому что взаимодействие происходит только между координатором и участниками, а сами участники между собой не взаимодействуют. Эта структура взаимодействия, являющаяся основой последующего обсуждения, более наглядно изображена на рис. 5.12.

Альтернатива – *линейный 2PC* (или *вложенный 2PC*), когда участники могут взаимодействовать между собой. Для целей взаимодействия узлы в системе могут быть логически упорядочены. Предположим, что узлы, участвующие в выполнении транзакции, пронумерованы 1, …, N, причем координатору присвоен номер 1. Протокол 2PC реализуется прямым взаимодействием между координатором (номер 1) и узлом N, во время которого завершается первая фаза, и обратным взаимодействием между узлом N и координатором, во время которого завершается вторая фаза. Стало быть, линейный 2PC работает следующим образом.

Координатор отправляет сообщение «prepare» участнику 2. Если участник 2 не готов зафиксировать транзакцию, он отправляет сообщение «vote-abort»

(VA) участнику 3, и в этот момент транзакция отменяется (односторонняя отмена участником 2). С другой стороны, если участник 2 согласен зафиксировать транзакцию, то он отправляет сообщение «vote-commit» (VC) участнику 3 и переходит в состояние READY. Как и в централизованной реализации 2PC, каждый узел заносит в журнал запись о своем решении, прежде чем отправлять сообщение следующему узлу. Этот процесс продолжается, пока сообщение «vote-commit» не дойдет до участника *N*. На этом первая фаза завершается. Если узел *N* решает зафиксировать транзакцию, то отправляет узлу (*N* – 1) сообщение «global-commit» (GC), в противном случае – сообщение «global-abort» (GA). В зависимости от полученного сообщения участники переходят в состояние COMMIT или ABORT и переправляют сообщение координатору.

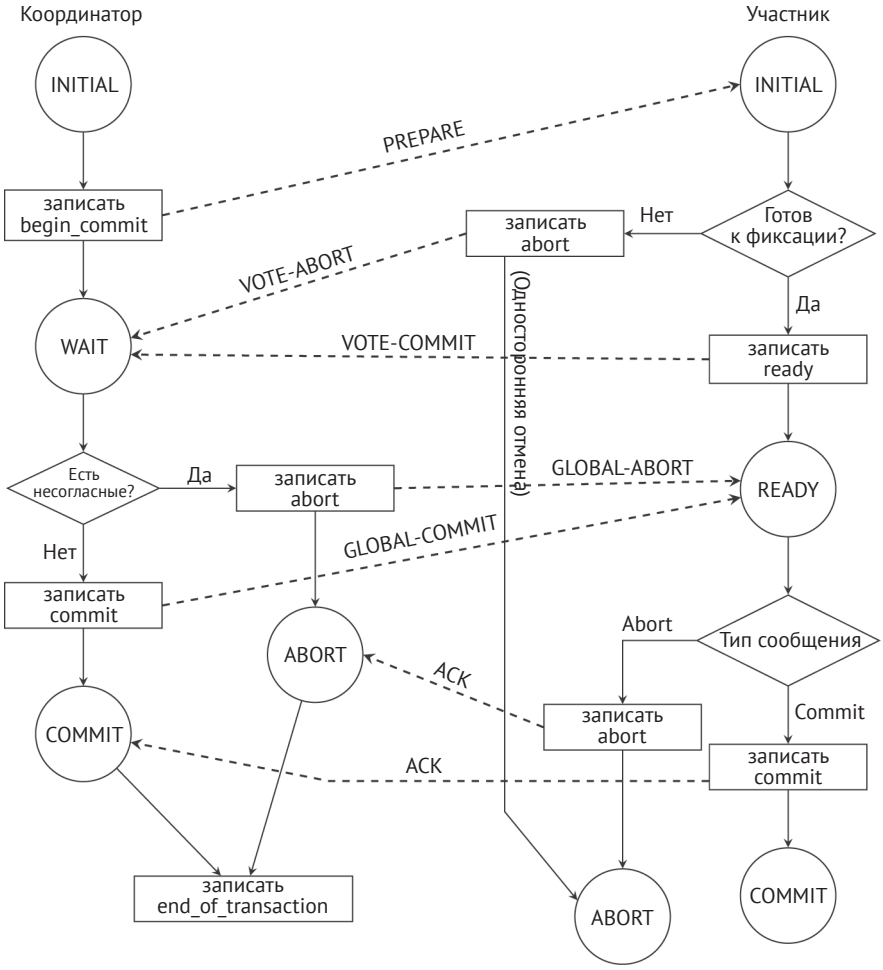


Рис. 5.11 ❖ Действия протокола 2PC

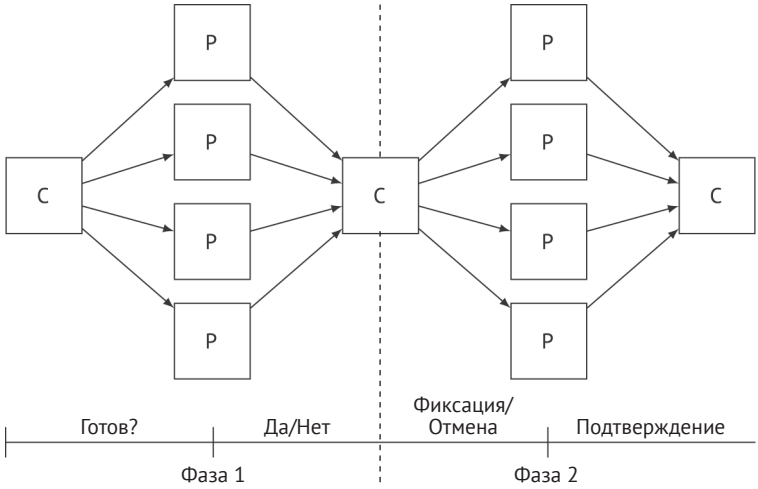


Рис. 5.12 ❖ Структура взаимодействия в линейном 2PC:
C – координатор; P – участник

В линейном протоколе 2PC, структура взаимодействия в котором показана на рис. 5.13, сообщений меньше, но зато нет никакого параллелизма. Поэтому время ответа в нем больше.

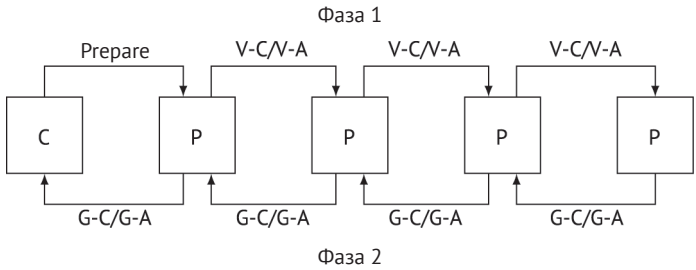


Рис. 5.13 ❖ Структура взаимодействия в линейном 2PC:
V-C – vote-commit; V-A – vote-abort; G-C – global-commit; G-A – global-abort

Еще одна популярная структура взаимодействия в реализации протокола 2PC подразумевает взаимодействие между всеми участниками на первой фазе, так что все они независимо приходят к решениям о том или ином завершении данной транзакции. В этой версии, называемой *распределенный 2PC*, во второй фазе вообще нет необходимости, т. к. участники могут прийти к решению самостоятельно. Работает она следующим образом. Координатор отправляет сообщение «prepare» всем участникам. Каждый участник посылает свое решение всем остальным участникам (и координатору) в сообщении «vote-commit» или «vote-abort». Каждый участник ждет сообщений от всех остальных участников и принимает решение о завершении по правилу глобальной фиксации. Очевидно, что вторая фаза протокола (когда кто-то отправляет сообщение global-abort или global-commit всем остальным) не

нужна, поскольку каждый участник пришел к этому решению независимо в конце первой фазы. Структура взаимодействия в распределенном варианте протокола показана на рис. 5.14.

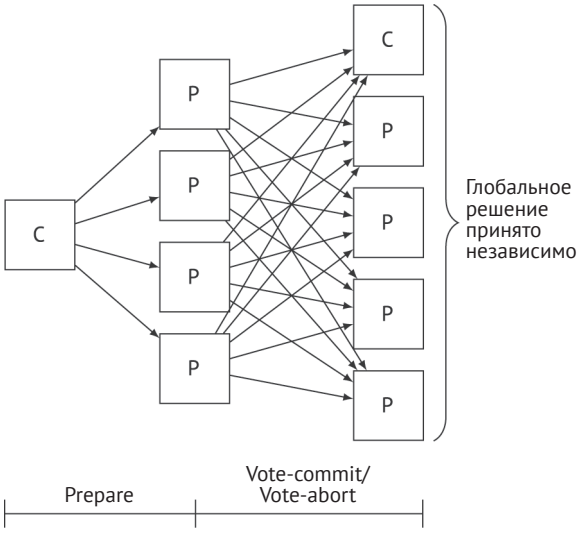


Рис. 5.14 ❖ Структура взаимодействия в распределенном 2PC

В линейной и распределенной реализации 2PC участник должен знать идентификатор следующего участника в линейном порядке (в случае линейного 2PC) или идентификаторы всех участников (в случае распределенного 2PC). Эту проблему можно решить, присоединив список участников к сообщению «prepare», которое отправляет координатор. Такая проблема не возникает в случае централизованного 2PC, поскольку координатор точно знает всех участников.

Алгоритмы централизованного выполнения протокола 2PC координатором и участниками приведены в алгоритмах 5.6 и 5.7 соответственно.

5.4.2. Варианты 2PC

Для улучшения производительности 2PC было предложено два варианта. Этого можно достичь уменьшением (1) количества сообщений, которыми обмениваются координатор и участники, и (2) количества записей в журнал. Соответствующие протоколы называются *предполагаемая отмена* и *предполагаемая фиксация*. Протокол предполагаемой отмены оптимизирован для обработки транзакций чтения и тех транзакций обновления, для которых некоторые процессы не производят обновления базы данных (так называемых транзакций частичного чтения). Протокол предполагаемой фиксации оптимизирован для обработки общих транзакций обновления. Мы кратко обсудим оба варианта.

Алгоритм 5.6. Координатор 2PC (2PC-C)

```
begin
    repeat
        ждать события event
        switch event do
            case Поступило сообщение do
                Обозначим поступившее сообщение msg
                switch msg do
                    case Commit do          {команда фиксации от планировщика}
                        поместить запись begin_commit в журнал
                        отправить сообщение «Prepared» всем участникам
                        взвести таймер
                    end case
                    case Vote-abort do  {один участник проголосовал за отмену;
                                                односторонняя отмена}
                        поместить запись abort в журнал
                        отправить сообщение «Global-abort» всем участникам
                        взвести таймер
                    end case
                    case Vote-commit do
                        обновить список ответивших участников
                        if ответили все участники then      {все проголосовали}
                            поместить запись commit в журнал
                            отправить сообщение «Global-commit» всем
                            участникам
                            взвести таймер
                        end if
                    end case
                    case Ack do
                        обновить список подтвердивших участников
                        if подтвердили все участники then
                            поместить запись end_of_transaction в журнал
                        else
                            отправить глобальное решение не ответившим
                            участникам
                        end if
                    end case
                end switch
            end case
            case Тайм-аут do
                выполнить протокол завершения
            end case
        end switch
    until бесконечно
end
```

Алгоритм 5.7. Участник 2PC (2PC-P)

```
begin
    repeat
        ждать события event
        switch event do
            case Поступило сообщение do
                Обозначим поступившее сообщение msg
                switch msg do
                    case Prepare do          {команда prepare от координатора}
                        if готов к фиксации then
                            поместить запись ready в журнал
                            отправить сообщение «Vote-commit» координатору
                            взвести таймер
                        end if
                        else                                {односторонняя отмена}
                            поместить запись abort в журнал
                            отправить сообщение «Vote-abort» координатору
                            отменить транзакцию
                        end if
                    end case
                    case Global-abort do
                            поместить запись abort в журнал
                            отменить транзакцию
                    end case
                    case Global-commit do
                        поместить запись commit в журнал
                        зафиксировать транзакцию
                    end case
                end switch
            end case
            case Тайм-аут do
                выполнить протокол завершения
            end case
        end switch
    until бесконечно
end
```

5.4.2.1. Протокол 2PC с предполагаемой отменой

В протоколе предполагаемой отмены, если готовый участник опрашивает координатора об исходе транзакции и не получает информации, ответом на запрос является отмена транзакции. Это работает, потому что в случае фиксации координатор не забывает о транзакции, пока все участники не пришлют подтверждение, и тем самым гарантируется, что они больше не будут наводить справки об этой транзакции.

Если используется такое соглашение, то, как легко видеть, координатор может забыть о транзакции, как только примет решение отменить ее. Он мо-

жет поместить запись abort в журнал и не ожидать, что участники подтвердят команду отмены. После записи abort координатор не обязан помещать еще и запись end_of_transaction.

Запись abort необязательна, потому что если узел выйдет из строя раньше, чем примет решение, а затем восстановится, то процедура восстановления проверит журнал, чтобы определить судьбу транзакции. Поскольку запись abort необязательна, процедура восстановления может не найти информации о транзакции, после чего обратится к координатору, который велит отменить транзакцию. По той же причине записи abort необязательны на стороне участников.

Поскольку протокол предполагаемой отмены избавляет от необходимости передавать некоторые сообщения между координатором и участниками в случае отмены транзакций, он считается более эффективным.

5.4.2.2. Протокол 2PC с предполагаемой фиксацией

Протокол 2PC с предполагаемой фиксацией основан на предположении, что если о транзакции нет никакой информации, то следует считать ее зафиксированной. Однако он не является в точности двойственным протоколу предполагаемой отмены, потому что это означало бы, что координатор забывает о транзакции сразу после решения зафиксировать ее, что записи commit (а также записи ready участников) необязательны и что команды фиксации не нужно подтверждать. Однако рассмотрим следующий сценарий. Координатор отправляет сообщения «prepare» и начинает собирать информацию, но выходит из строя, не успев собрать всю и принять решение. В таком случае участники дождутся тайм-аута, а затем препоручат транзакцию своим процедурам восстановления. Поскольку о транзакции нет информации, процедуры восстановления каждого участника зафиксируют ее. С другой стороны, координатор после восстановления отменит транзакцию – налицо несогласованность.

В протоколе предполагаемой фиксации эта проблема решается следующим образом. Координатор, перед тем как отправлять сообщение «prepare», помещает в журнал запись collecting, которая содержит имена всех участников выполнения данной транзакции. Затем координатор переходит в состояние COLLECTING, отправляет сообщение «prepare» и переходит в состояние WAIT. Получив сообщение «prepare», участники решают, что они хотят сделать с транзакцией, помещают в журнал соответствующую запись (abort или ready) и отвечают сообщением «vote-abort» или «vote-commit». Получив решения участников, координатор принимает решение об отмене или фиксации транзакции. Решив отменить, координатор помещает запись abort, переходит в состояние ABORT и отправляет сообщение «global-abort». Решив зафиксировать, координатор помещает запись commit, отправляет сообщение «global-commit» и забывает о транзакции. Получив сообщение «global-commit», участники помещают в свои журналы запись commit и обновляют базу данных. Получив же сообщение «global-abort», они помещают запись abort и подтверждают. Координатор, получив подтверждение отмены, помещает в журнал запись end_of_transaction и забывает о транзакции.

5.4.3. Обработка отказов узлов

В этом разделе мы рассмотрим отказы узлов сети. Наша цель – разработать протоколы неблокирующего завершения и независимого восстановления. Как уже было отмечено, из существования протоколов независимого восстановления следует существование протоколов неблокирующего завершения. Однако мы будем обсуждать оба аспекта по отдельности. Заметим также, что ниже рассматривается только стандартный протокол 2PC, а не два его представленных выше варианта.

Сначала очертим границы существования протоколов неблокирующего завершения и независимого восстановления при наличии отказов узлов. Доказано, что такие протоколы существуют, если отказывает один узел. Но в случае отказа нескольких узлов перспективы не столь радужны. Отрицательный результат означает, что невозможно спроектировать протоколы независимого восстановления (а значит, и протоколы неблокирующего завершения), если отказывает несколько узлов. Сначала мы разработаем протоколы завершения и восстановления для алгоритма 2PC и покажем, что 2PC принципиально блокирующий. Затем перейдем к разработке протоколов атомарной фиксации, которые являются неблокирующими в случае отказа одного узла.

5.4.3.1. Протоколы завершения и восстановления для 2PC

Протоколы завершения

Протоколы завершения обслуживают тайм-ауты для процессов координатора и участника. Тайм-аут в узле-приемнике возникает, когда тот не получает ожидаемое сообщение от узла-источника в течение предопределенного периода времени. В этом разделе мы будем считать, что это происходит из-за отказа узла-источника.

Метод обработки тайм-аутов зависит от момента и типа отказа. Поэтому мы будем рассматривать отказы в различных точках выполнения протокола 2PC и обращаться к диаграмме переходов состояний на рис. 5.10.

Тайм-ауты координатора

Тайм-аут координатора может случиться в трех состояниях: WAIT, COMMIT и ABORT. Два последних обрабатываются одинаково. Поэтому нужно рассмотреть только два случая:

1) *тайм-аут в состоянии WAIT*. В этом состоянии координатор ждет локальных решений участников. Координатор не может в одностороннем порядке зафиксировать транзакцию, поскольку не удовлетворено правило глобальной фиксации. Однако он может принять решение о глобальной отмене транзакции, и в таком случае он помещает запись `abort` в журнал и отправляет сообщение «global-abort» всем участникам;
2) *тайм-аут в состояниях COMMIT и ABORT*. В этом случае координатор не уверен, завершены ли процедуры фиксации или отмены локальными диспетчерами восстановления в узлах-участниках. Поэтому коор-

динатор повторно отправляет сообщения «global-commit» или «global-abort» узлам, которые еще не ответили, и ждет подтверждения.

Тайм-ауты участника

Тайм-аут участника[1] может случиться в двух состояниях: INITIAL и READY. Рассмотрим оба случая:

1) *тайм-аут в состоянии INITIAL*. В этом состоянии участник ждет сообщения «prepare». Раз оно не пришло, значит, координатор, по-видимому, вышел из строя. Участник может в одностороннем порядке отменить транзакцию после тайм-аута. Если сообщение «prepare» придет позже, то его можно будет обработать одним из двух способов. Либо участник проверит свой журнал, найдет запись `abort` и ответит сообщением «vote-abort», либо попросту проигнорирует сообщение «prepare». В последнем случае координатор столкнется с тайм-аутом в состоянии WAIT и будет действовать, как описано выше;
2) *тайм-аут в состоянии READY*. В этом состоянии участник проголосовал за фиксацию транзакции, но не знает о глобальном решении координатора. Участник не может принять решение в одностороннем порядке. Поскольку он находится в состоянии READY, то наверняка уже проголосовал за фиксацию. Поэтому он не может изменить свое решение и односторонне отменить транзакцию. Но и односторонне зафиксировать ее он тоже не может, потому что другой участник мог проголосовать за отмену. Поэтому участник остается заблокированным, пока не узнает от кого-то (от координатора или другого участника) о судьбе транзакции.

Рассмотрим централизованную структуру взаимодействия, когда участники не могут взаимодействовать друг с другом. В этом случае участник, пытающийся завершить транзакцию, должен спросить координатора о его решении и ждать ответа. Если координатор вышел из строя, то участник останется заблокированным. Это нежелательно.

Если участники могут взаимодействовать друг с другом, то можно разработать в большей степени распределенный протокол завершения. Участник, столкнувшийся с тайм-аутом, может просто попросить всех остальных участников помочь ему в принятии решения. В предположении, что тайм-аут испытал участник P_i, каждый из остальных участников (P_j) отвечает в соответствии со следующими правилами:

1) P_j находится в состоянии INITIAL. Это означает, что P_j еще не проголосовал и, быть может, даже не получил сообщения «prepare». Поэтому он может в одностороннем порядке отменить транзакцию и ответить P_i сообщением «vote-abort»;
2) P_j находится в состоянии READY. В этом состоянии P_j проголосовал за фиксацию транзакции, но еще не получил информации о глобальном решении. Поэтому он не может помочь P_i в завершении транзакции;

---

[1] В некоторых обсуждениях протокола 2PC предполагается, что участники не используют таймеры и тайм-аута у них не бывает. Однако реализация протоколов обработки тайм-аутов в узлах-участниках решает ряд неприятных проблем и может ускорить процесс фиксации. Поэтому мы рассматриваем этот более общий случай.

3) P_j находится в состоянии ABORT или COMMIT. В этих состояниях P_j либо принял одностороннее решение отменить транзакцию, либо получил решение координатора о глобальном завершении. Поэтому он может отправить P_i сообщение «vote-commit» или «vote-abort».

Рассмотрим, как участник, столкнувшийся с тайм-аутом (P_i), может интерпретировать эти ответы. Возможны следующие случаи:

1) P_i получает сообщения «vote-abort» от всех P_j. Это означает, что ни один участник еще не проголосовал, но они решили отменить транзакцию в одностороннем порядке. При таких условиях P_i может перейти к отмене транзакции;
2) P_i получает сообщения «vote-abort» от некоторых P_j, но другие участники сообщают, что находятся в состоянии READY. В этом случае P_i все равно может продолжить и отменить транзакцию, поскольку, согласно правилу глобальной фиксации, транзакция не может быть зафиксирована и в конечном итоге будет отменена;
3) P_i получает от всех P_j уведомления о том, что они находятся в состоянии READY. В таком случае ни один участник не имеет достаточной информации о судьбе транзакции, чтобы правильно завершить ее;
4) P_i получает сообщения «global-abort» или «global-commit» от всех P_j. В этом случае все остальные участники получили решение координатора. Поэтому P_i может завершить транзакцию в соответствии с сообщениями, полученными от других участников. Кстати, отметим, что не может быть так, что одни участники P_j ответят «global-abort», а другие – «global-commit», потому что это противоречило бы протоколу 2PC;
5) P_i получает сообщения «global-abort» или «global-commit» от некоторых P_j, тогда как остальные сообщают, что находятся в состоянии READY. Это означает, что некоторые узлы получили решение координатора, а остальные еще ждут его. В таком случае P_i может продолжить, как в случае 4 выше.

Этими пятью случаями исчерпываются все варианты, которые должен обрабатывать протокол завершения. Например, необязательно рассматривать случай, когда один участник посылает сообщение «vote-abort», а другой «global-commit». В протоколе 2PC такого произойти не может. Во время выполнения 2PC никакой процесс (участник или координатор) не может находиться в состоянии, отстоящем более чем на одно состояние от любого другого процесса. Так, если участник находится в состоянии INITIAL, то все остальные участники находятся в состоянии INITIAL или READY. Аналогично координатор находится в состоянии INITIAL или WAIT. Поэтому говорят, что в протоколе 2PC все процессы *синхронны с точностью до одного перехода состояний*.

Заметим, что в случае 3 процессы-участники остаются заблокированными, поскольку не могут завершить транзакцию. При определенных обстоятельствах эту блокировку можно преодолеть. Если в ходе завершения все участники придут к выводу, что узел координатора вышел из строя, то они могут выбрать нового координатора, который перезапустит процесс фиксации. Есть разные способы выбора координатора. Можно либо определить

полный порядок на множестве всех узлов и выбрать следующий узел по порядку, либо провести выборы среди участников. Но это не сработает, если два узла – участник и координатор – отказали. В таком случае может случиться, что участник в отказавшем узле получил решение координатора и соответственно завершил транзакцию. Это решение неизвестно другим участникам, поэтому, если они выберут нового координатора и продолжат, то возникнет опасность, что они решат завершить транзакцию иначе, чем участник в отказавшем узле. Ясно, что невозможно придумать протоколы завершения для 2PC, которые гарантировали бы неблокирующее завершение. Поэтому протокол 2PC является блокирующим. Формально этот протокол блокирующий, потому что на рис. 5.10 существует состояние, соседствующее одновременно с состояниями COMMIT и ABORT, а когда координатор выходит из строя, участники находятся в состоянии READY. Поэтому невозможно определить, собирался ли координатор перейти в состояние ABORT или COMMIT, до того как он восстановится. Эту ситуацию разрешает протокол 3PC (трехфазной фиксации), в который добавлено новое состояние PRECOMMIT между состояниями WAIT и COMMIT, чтобы предотвратить блокировку в случае отказа координатора.

Поскольку при разработке протокола 2PC в алгоритмах 5.6 и 5.7 мы предполагали централизованную структуру взаимодействия, продолжим придерживаться той же линии при разработке протоколов завершения. Код, который следует включить в секцию тайм-аута алгоритмов координатора и участника, показан соответственно в алгоритмах 5.8 и 5.9.

Алгоритм 5.8. Завершение координатора 2PC

```
begin
    if в состоянии WAIT then          {координатор находится в состоянии ABORT}
        поместить запись write в журнал
        отправить сообщение «global-abort» всем участникам
    else                              {координатор находится в состоянии COMMIT}
        проверить последнюю запись в журнале
        if последняя запись в журнале = abort then
            отправить сообщение «global-abort» всем участникам, которые еще
            не ответили
        else
            отправить сообщение «global-commit» всем участникам, которые
            еще не ответили
        end if
    end if
    взвести таймер
end
```

Протоколы восстановления

В предыдущем разделе мы обсудили, как в протоколе 2PC обрабатываются ошибки с точки зрения сохранивших работоспособность узлов. А в этом разделе сменим угол зрения: нас будут интересовать протоколы, которыми координатор или участник может воспользоваться, чтобы восстановить свое

состояние после отказа и перезапуска узла. Напомним, что мы хотим, чтобы эти протоколы были независимыми. Однако в общем случае невозможно спроектировать протоколы, которые гарантировали бы независимое восстановление, сохранив в то же время атомарность распределенных транзакций. Это неудивительно, принимая во внимание тот факт, что протоколы завершения для 2PC принципиально блокирующие.

Алгоритм 5.9. Завершение участника 2PC

```
begin
    if в состоянии INITIAL then
        поместить запись write в журнал
    else
        отправить сообщение «vote-commit» координатору
        переустановить таймер
    end if
end
```

Далее мы снова будем использовать диаграмму переходов состояний на рис. 5.10. Кроме того, сделаем два предположения относительно толкования диаграммы: (1) комбинированное действие – запись в журнал и отправка сообщения – считается атомарным и (2) переход состояний происходит после передачи ответного сообщения. Например, если координатор находится в состоянии WAIT, значит, он успешно поместил запись `begin_commit` в свой журнал и успешно передал команду «prepare». Но при этом мы ничего не знаем о том, успешно ли завершилась передача сообщения. Возможно, сообщение «prepare» не дошло до участников из-за ошибок связи, и этот вопрос мы обсудим отдельно. Конечно, первое предложение, касающееся атомарности, нереалистично. Но оно упрощает обсуждение фундаментальных отказов. В конце этого раздела мы покажем, что другие ошибки, которые могут возникнуть при ослаблении этого предположения, можно обработать как комбинацию фундаментальных отказов.

Отказы узла координатора

Возможны следующие случаи:

1) *координатор вышел из строя в состоянии INITIAL*. Это произошло перед тем, как координатор инициировал процедуру фиксации. Поэтому он начнет процесс фиксации после восстановления;
2) *координатор вышел из строя в состоянии WAIT*. В этом случае координатор успел послать команду «prepare». После восстановления координатор перезапустит процесс фиксации этой транзакции с самого начала, отправив сообщение «prepare» еще раз;
3) *координатор вышел из строя в состоянии COMMIT или ABORT*. В этом случае координатор уже проинформировал участников о своем решении и завершил транзакцию. Поэтому после восстановления ему не нужно делать ничего, если все подтверждения были получены. В противном случае включается в дело протокол завершения.

Отказы узла участника

Нужно рассмотреть три случая:

1) *участник вышел из строя в состоянии INITIAL*. После восстановления участник должен отменить транзакцию в одностороннем порядке. Посмотрим, почему так можно сделать. Заметим, что координатор будет находиться в состоянии INITIAL или WAIT по отношению к этой транзакции. Если он находится в состоянии INITIAL, то пошлет сообщение «prepare» и перейдет в состояние WAIT. Из-за отказа узла участника координатор не получит его решения и столкнется с тайм-аутом в этом состоянии. Мы уже рассматривали, как координатор обрабатывает тайм-ауты в состоянии WAIT, глобально отменяя транзакцию;
2) *участник вышел из строя в состоянии READY*. В этом случае координатор уже был проинформирован об утвердительном решении отказавшего сайта по поводу транзакции перед его выходом из строя. После восстановления участник в отказавшем узле сможет рассматривать эту ситуацию как тайм-аут в состоянии READY и препоручить незавершенную транзакцию заботам протокола завершения;
3) *участник вышел из строя в состоянии ABORT или COMMIT*. Это завершающие состояния, поэтому после восстановления участнику ничего делать не нужно.

Дополнительные случаи

Теперь посмотрим, какие случаи могут возникнуть, если ослабить предположение об атомарности записи в журнал и отправки сообщения. В частности, предположим, что отказ узла может произойти после того, как координатор или участник поместил запись в журнал, но до того, как он отправил сообщение. Эта ситуация иллюстрируется на рис. 5.11.

1. *Координатор вышел из строя после записи* `begin_commit` *в журнал, но до отправки команды «prepare»*. Координатор должен отреагировать на это как на отказ в состоянии WAIT (случай 2 при рассмотрении отказов координатора выше) и после восстановления отправить команду «prepare».
2. *Участник вышел из строя после записи* `ready` *в журнал, но до отправки сообщения «vote-commit»*. Отказавший участник рассматривает это как случай 2 в обсуждении отказов участника выше.
3. *Участник вышел из строя после записи* `abort` *в журнал, но до отправки сообщения «vote-abort»*. Это единственная ситуация, которая не встречалась при разборе основных случаев. Но после восстановления участнику ничего не нужно делать. Координатор находится в состоянии WAIT и испытает тайм-аут. Протокол завершения координатора в этом состоянии глобально отменяет транзакцию.
4. *Координатор вышел из строя после помещения в журнал записи об окончательном решении* (`Abort` или `Commit`), но до отправки сообщения «global-abort» или «global-commit» участникам. Координатор рассматривает это как свой случай 3, а участники – как тайм-аут в состоянии READY.

5. *Участник вышел из строя после записи* `abort` *или* `commit` *в журнал, но до отправки подтверждения координатору.* Участник может рассматривать эту ситуацию как свой случай 3, а координатор – как тайм-аут в состоянии COMMIT или ABORT.

5.4.3.2. Протокол трехфазной фиксации

Как было отмечено выше, блокирующие протоколы фиксации нежелательны. Протокол трехфазной фиксации (3PC) разработан как неблокирующий протокол для случая, когда возможны только отказы узлов. В случае отказа сети ситуация усложняется.

3PC интересен с алгоритмической точки зрения, но сопровождается более высокими накладными расходами на коммуникацию в виде дополнительных задержек, потому что включает три раунда обмена сообщениями и синхронную запись в постоянный журнал. Поэтому в реальных системах он не встречается – даже 2PC критикуют за высокие задержки из-за последовательных участков, на которых производится синхронная запись в журнал. Поэтому мы кратко опишем этот подход без подробного анализа.

Сначала рассмотрим необходимые и достаточные условия для проектирования неблокирующих протоколов атомарной фиксации. Протокол фиксации, синхронный в пределах одного перехода состояний, является неблокирующим тогда и только тогда, когда его диаграмма переходов состояний не содержит:

1) ни одного состояния, «соседнего» одновременно с состояниями COMMIT и ABORT;
2) ни одного не допускающего фиксации состояния, «соседнего» с состоянием COMMIT.

Здесь *соседний* означает, что из одного состояния можно попасть в другое за один переход.

Рассмотрим состояние COMMIT в протоколе 2PC (см. рис. 5.10). Если какой-нибудь процесс находится в этом состоянии, то мы точно знаем, что все участники проголосовали за фиксацию транзакции. Такие состояния называются *допускающими фиксацию*. Но в протоколе 2PC есть также состояния, *не допускающие фиксации*. Нас особенно интересует состояние READY, которое не допускает фиксации, потому что из существования процесса в этом состоянии не следует, что все процессы проголосовали за фиксацию транзакции.

Очевидно, что состояние WAIT координатора и состояние READY участника в протоколе 2PC нарушают сформулированные выше условия неблокируемости. Поэтому можно было бы внести следующее изменение в протокол 2PC, чтобы удовлетворить эти условия и превратить протокол в неблокирующий.

Добавим между WAIT (и READY) и COMMIT еще одно буферное состояние, в котором процесс готов к фиксации (если таково окончательное решение), но еще не зафиксировал транзакцию. Диаграммы переходов состояний координатора и участника для этого протокола показаны на рис. 5.15. Этот протокол называется трехфазной фиксацией (3PC), потому что на пути из INITIAL в COMMIT расположены три состояния. Выполнение протокола меж-

ду координатором и одним участником показано на рис. 5.16. Заметим, что он отличается от рис. 5.11 только добавлением состояния PRECOMMIT. Кроме того, в протоколе 3PC все состояния синхронны в пределах одного перехода состояний. Поэтому к нему применимы условия неблокируемости.

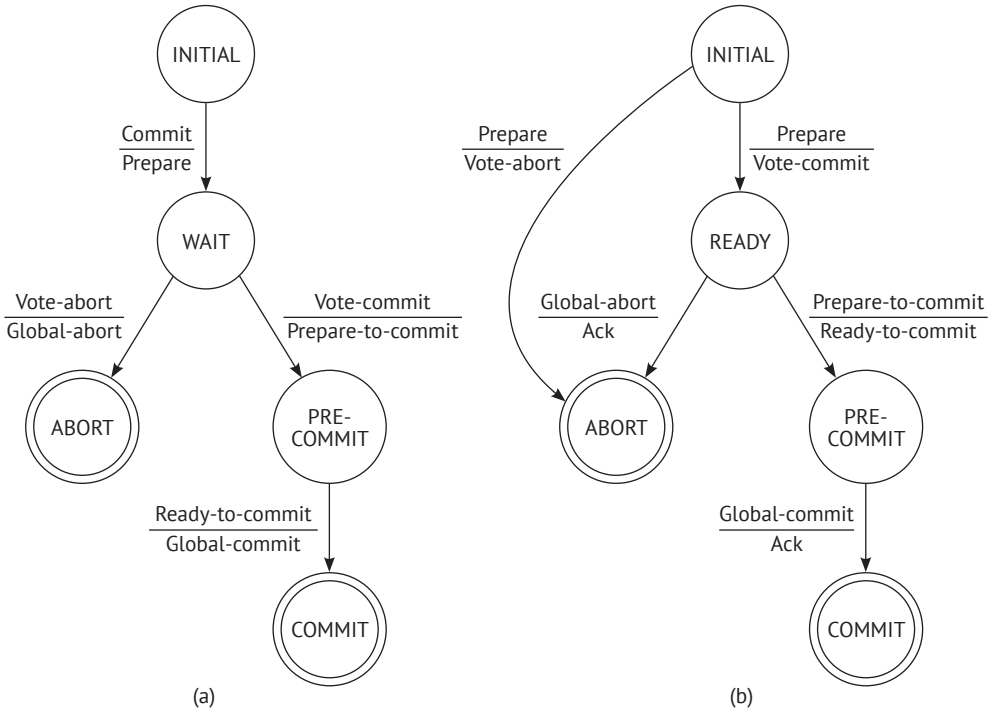


Рис. 5.15 ❖ Переходы состояний в протоколе 3PC: (a) состояния координатора; (b) состояния участника

5.4.4. Разделение сети

В этом разделе мы рассмотрим, как разделение сети обрабатывается в протоколах атомарной фиксации, описанных в предыдущем разделе. Разделение сети – это последствие отказов линий связи; в зависимости от реализации сети оно может привести к потере сообщений. Разделение называется *простым*, если сеть распалась только на две части, в противном случае разделение называется *множественным*.

Протоколы завершения для разделения сети решают задачу завершения транзакций, которые были активны в каждой части в момент разделения. Если возможно разработать неблокирующие протоколы для завершения таких транзакций, то узлы в каждой части сети смогут прийти к решению о завершении (данной транзакции), согласованные с узлами в других частях. Отсюда следовало бы, что узлы в каждой части сети могут продолжать выполнение транзакций, несмотря на разделение.

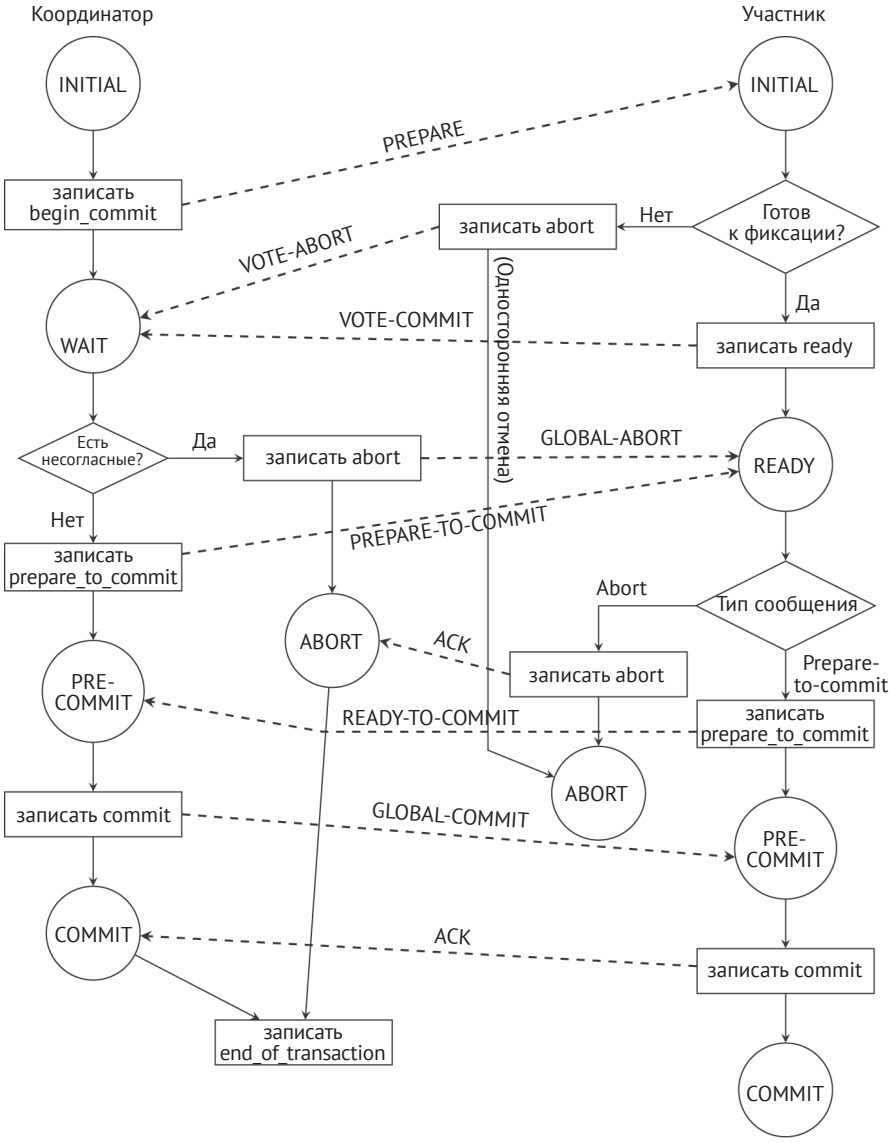


Рис. 5.16 ❖ Действия протокола 3PC

К сожалению, в общем случае невозможно спроектировать неблокирующие протоколы завершения при наличии разделения сети. Напомним, что наши предположения о надежности сети связи минимальны. Если сообщение не может быть доставлено, то оно просто теряется. В таком случае можно доказать, что не существует неблокирующего протокола атомарной фиксации, устойчивого к разделению сети. Это крайне неутешительный результат, потому что он также означает, что коль скоро разделение сети произошло, то мы не сможем продолжать нормальную работу во всех разделах, что

ограничивает доступность распределенной системы базы данных в целом. Но есть и положительный результат – оказывается, можно спроектировать неблокирующие протоколы атомарной фиксации, устойчивые к простым разделениям. К сожалению, при наличии множественного разделения таких протоколов не существует.

Далее в этом разделе мы обсудим ряд протоколов для решения проблемы разделения сети в нереплицированных базах данных. В случае реплицированных баз задача выглядит совершенно иначе, и мы обсудим ее в следующей главе.

При разделении сети в нереплицированной базе данных наша главная проблема – завершение транзакций, активных в момент разделения. Любая новая транзакция, которая пытается обратиться к элементу данных, хранящемуся в другой части базы, попросту блокируется и должна будет ждать восстановления сети. Конкурентные обращения к элементам данных внутри одной части можно обработать с помощью алгоритма управления конкурентностью. Поэтому важно только обеспечить правильное завершение транзакций. Короче говоря, проблема разделения сети решается протоколом фиксации, а точнее протоколами завершения и восстановления.

Отсутствие неблокирующих протоколов, которые гарантировали бы атомарную фиксацию распределенных транзакций, ведет к важному проектному решению. Мы можем либо разрешить всем частям продолжать обычное функционирование, приняв тот факт, что согласованность базы данных может быть нарушена, либо гарантировать согласованность, применяя стратегии, которые разрешали бы операции в одной части, пока узлы в прочих частях остаются блокированными. Этот выбор лежит в основе классификации стратегий обработки разделения на *пессимистические* и *оптимистические*. В пессимистических стратегиях на первое место ставится согласованность базы данных, поэтому транзакциям не разрешено выполняться в какой-либо части, если нет гарантии, что согласованность можно поддержать. С другой стороны, в оптимистических стратегиях во главу угла ставится доступность базы данных, даже если это ведет к несогласованности.

Второй фактор связан с критерием корректности. Если в качестве фундаментального критерия корректности используется сериализуемость, то стратегия называется *синтаксической*, поскольку в теории сериализуемости используется только синтаксическая информация. Если же применяется более абстрактный критерий корректности, зависящий от семантики транзакций в базе данных, то стратегия называется *семантической*.

Сохраняя приверженность принятому в этой книге критерию корректности (сериализуемость), мы в этом разделе будем рассматривать только синтаксические подходы. В следующих двух разделах описаны различные синтаксические стратегии для нереплицированных баз данных.

Все известные протоколы завершения, работающие в условиях разделения сети для нереплицированных баз данных, пессимистические. Поскольку в пессимистических подходах на первом месте стоит поддержание согласованности базы данных, фундаментальная проблема, которую нам предстоит решить, – в каких частях базы можно продолжать нормальную работу. Мы рассмотрим два решения.

5.4.4.1. Централизованные протоколы

Централизованные протоколы завершения основаны на централизованных алгоритмах управления конкурентностью, рассмотренных в разделе 5.2. В этом случае имеет смысл разрешить работу в части, содержащей центральный узел, потому что он управляет таблицами блокировок.

Методы главного узла централизованы относительно каждого элемента данных. В этом случае для некоторых запросов может быть несколько работоспособных частей. Для любого запроса транзакцию может выполнять только та часть, которая содержит главный узел элементов данных, находящихся во множестве записи этой транзакции.

Оба этих подхода просты и могли бы хорошо работать, но они зависят от конкретного механизма управления конкурентностью. Кроме того, ожидается, что каждый узел сможет отличить разделение сети от отказа узлов. Это необходимо, потому что участники, исполняющие протокол фиксации, по-разному реагируют на различные типы отказов. К сожалению, в общем случае это невозможно.

5.4.4.2. Протоколы на основе голосования

Голосование также можно использовать для управления конкурентным доступом к данным. В качестве метода управления конкурентностью для полностью реплицированных баз данных было предложено прямолинейное мажоритарное голосование. Фундаментальная идея заключается в том, что транзакция выполняется, если большинство узлов проголосовало за ее выполнение.

Идея мажоритарного голосования была обобщена на голосование с *кворумом*. Такое голосование можно использовать как метод управления репликами (обсуждается в следующей главе), а также как метод фиксации, гарантирующий атомарность транзакции в условиях разделения сети. В случае нереплицированных баз данных принцип голосования интегрируется в протоколы фиксации. Мы представим одно конкретное предложение в этом духе.

Каждому узлу в системе назначается голос V_i. Предположим, что общее число голосов в системе равно V, а кворумы для отмены и фиксации равны V_a и V_c соответственно. Тогда в реализации протокола фиксации должны быть соблюдены следующие правила:

1) $V_a + V_c > V$, где $0 \leqslant V_a, V_c \leqslant V$;
2) перед фиксацией транзакции должен быть набран кворум фиксации V_c;
3) перед отменой транзакции должен быть набран кворум отмены V_a.

Первое правило гарантирует, что транзакция не будет одновременно зафиксирована и отменена. Следующие два правила регламентируют количество голосов, которые должна получить транзакция для завершения тем или иным способом. Интеграцию кворумов в протоколы фиксации оставляем читателю в качестве упражнения.

5.4.5. Протокол достижения консенсуса Paxos

До сих пор мы изучали протоколы 2PC для достижения согласия между диспетчерами транзакций по поводу атомарного выполнения распределенной транзакции и обнаружили, что они обладают нежелательным свойством блокировки в случае, когда отказывает координатор и еще один участник. Мы обсудили, как преодолеть этот недостаток с помощью протокола 3PC, который, однако, обходится дорого и неустойчив к разделению сети. При рассмотрении вопроса о разделении сети мы упомянули голосование как способ определить часть системы, в которой находится «большинство» диспетчеров транзакций, и завершить транзакцию в этой части. Это может показаться частичным решением фундаментальной проблемы нахождения отказоустойчивых механизмов достижения согласия (консенсуса) между диспетчерами транзакций по вопросу о судьбе рассматриваемой транзакции. Как выясняется, достижение консенсуса между узлами – общая задача распределенных вычислений, известная под названием *распределенный консенсус*. Для ее решения предложен целый ряд алгоритмов; в этом разделе мы рассмотрим семейство алгоритмов Paxos, а остальные упомянем в библиографических замечаниях.

Сначала обсудим Paxos в общей постановке, в которой он и был определен первоначально, а затем разберем, как его применить в протоколах фиксации. Общий алгоритм предназначен для достижения консенсуса между узлами по вопросу о значении некоторой переменной (или решения). Важно, что консенсус считается достигнутым, если за значение проголосовало большинство узлов, но, быть может, не все. Таким образом, сколько-то узлов могут быть неработоспособны, но коль скоро большинство существует, консенсуса можно достичь. В алгоритме определены три роли: *заявитель* рекомендует значение переменной, *акцептор* решает, принять рекомендованное значение или нет, а *ученик* узнает о согласованном значении, спрашивая одного из акцепторов (или же акцептор сам сообщает ему значение). Заметим, что все эти роли могут быть размещены в одном узле, но ни в каком узле не может существовать более одного экземпляра каждой роли. Ученики не особенно важны, поэтому в нашем изложении мы не будем останавливаться на них сколько-нибудь подробно.

Протокол Paxos прост, если имеется всего один заявитель, и работает он, как протокол 2PC: в первом раунде заявитель предлагает значение переменной, а акцепторы отправляют свои ответы (принять или не принять). Если заявитель получает согласие большинства акцепторов, то решает, что это значение и станет значением переменной, после чего уведомляет акцепторов, которые запоминают это значение в качестве окончательного. Ученик может в любой момент запросить у акцептора значение переменной и получит самое последнее.

Конечно, реальность не так проста, и протокол Paxos должен как-то справляться со следующими трудностями:

1) поскольку это, по определению, распределенный протокол достижения консенсуса, несколько заявителей могут предложить значение одной

и той же переменной. Поэтому акцептор должен выбрать одно из предложенных значений;

2) если имеется несколько предложений, голоса могут разделиться между ними, так что ни одно значение не наберет большинства;
3) может случиться, что некоторые акцепторы выйдут из строя, после того как примут значение. Если оставшиеся акцепторы, принявшие значение, не составляют большинства, возникает проблема.

Первую проблему Paxos решает с помощью баллотировочного числа, так чтобы акцепторы могли различать предложения (см. ниже). Вторую проблему можно решить, проведя несколько раундов голосования, – если ни одно предложение не набирает большинства, проводится второй раунд, и так повторяется, пока какое-то значение не соберет большинства голосов. В некоторых случаях итераций может быть много, что приводит к снижению производительности. Paxos решает эту проблему, назначая *лидера*, которому каждый заявитель отправляет предлагаемое значение. Затем лидер выбирает одно значение каждой переменной и пытается получить большинство. Это идет несколько вразрез с распределенной природой протокола достижения консенсуса. В различных раундах выполнения протокола лидер может меняться. Третья проблема более серьезна. Можно было бы трактовать ее так же, как вторую, и организовать новый раунд. Но беда в том, что некоторые ученики могли запомнить значение, полученное от акцепторов в предыдущем раунде, и если в новом раунде будет выбрано другое значение, то возникнет рассогласование. В Paxos и эта проблема решается с помощью баллотировочных чисел.

Мы представим шаги «базового Paxos», цель которого – определить значение одной переменной (поэтому далее имя переменной опускается). В базовом Paxos также упрощено определение баллотировочных чисел: в данном случае такие числа должны быть только уникальными и монотонно возрастающими в пределах каждого заявителя; не делается никаких попыток сделать их глобально уникальными, потому что консенсус считается достигнутым, когда большинство акцепторов согласятся с *каким-то* значением переменной вне зависимости от того, кто его предложил. Ниже описана работа базового Paxos в отсутствие ошибок.

S1. Заявитель, желающий начать поиск консенсуса, отправляет всем акцепторам сообщение «prepare» со своим баллотировочным числом [prepare(*bal*)].

S2. Каждый акцептор, получивший сообщение «prepare», выполняет следующие действия:
- если он не получал никаких предложений раньше;
- то он записывает prepare(*bal*) в свой журнал и отвечает сообщением ack(*bal*);
- иначе если *bal* > больше любого баллотировочного числа, которое он получал от любого заявителя раньше;
- то он записывает prepare(*bal*) в свой журнал и отвечает сообщением, содержащим баллотировочное число (*bal′*) и значение (*val′*) самого большого предложенного числа, которое он принял до этого: ack(*bal*, *bal′*, *val′*);

- иначе он игнорирует сообщение «prepare».

S3. Получив сообщение ack от акцептора, заявитель записывает его в журнал.

S4. Получив сообщения ack от большинства акцепторов (т. е. набрав кворум, достаточный для консенсуса), заявитель отправляет акцепторам сообщение accept(*nbal*, *val*), где *nbal* – баллотировочное число, а *val* – значение, подлежащее приемке. Величины *nbal* и *val* определяются следующим образом:
   - если все полученные заявителем сообщения ack говорят о том, что ни один из акцепторов ранее не принимал значения,
   - то предлагаемое значение *val* устанавливается равным тому, которое заявитель хотел предложить с самого начала, а $nbal \leftarrow bal$,
   - иначе *val* устанавливается равным значению val' в ответном сообщении ack с наибольшим значением bal', а $nbal \leftarrow bal$ (так что процесс сходится к значению с наибольшим баллотировочным числом);
   - заявитель отправляет всем акцепторам сообщение accept(*bal*, *val*), где *val* – предлагаемое значение.

S5. Каждый акцептор, получив сообщение accept(*nbal*, *val*), выполняет следующие действия:
   - если *nbal* = ack.*bal* (т. е. баллотировочное число в сообщении accept совпадает с тем, что он обещал раньше),
   - то он записывает accept(*nbal, val*),
   - иначе игнорирует сообщение.

По поводу этого протокола следует сделать ряд замечаний. Во-первых, на шаге S2 акцептор игнорирует сообщение prepare, если уже получал сообщение prepare с баллотировочным числом большим, чем только что полученное. Хотя протокол в этом случае работает корректно, заявитель может продолжить взаимодействие с этим акцептором позже (например, на шаге S5), а этого можно избежать, если акцептор отправит отрицательное подтверждение, чтобы заявитель мог исключить его из рассмотрения в будущем. Во-вторых, когда акцептор подтверждает сообщение prepare, он подтверждает и то, что заявитель является лидером Paxos в этом раунде. Поэтому в некотором смысле выбор лидера производится в самом протоколе. Однако для решения второй из вышеупомянутых проблем можно иметь назначенного лидера, который затем инициирует раунды. Если избрана такая реализация, то выдвинутый лидер может решить, какое предложение поставить на голосование, если их несколько. Наконец, поскольку протокол продвигается вперед, если доступно большинство участников (узлов), то в системе с N узлами он может вынести $N/2 - 1$ одновременных отказов узлов.

Теперь кратко проанализируем, как в Paxos обрабатываются ошибки. Простейший случай – когда несколько акцепторов выходят из строя, но кворум для принятия решения все же присутствует. В этом случае протокол работает как обычно. Ситуация, когда из строя вышло столько акцепторов, что кворум не набирается, естественно обрабатывается проведением нового голосования (или нескольких голосований), когда кворум соберется. Самый трудный случай для всех алгоритмов достижения консенсуса – отказ заявителя (являющегося также лидером). В этом случае Paxos выбирает нового

лидера с помощью того или иного механизма (в литературе есть немало предложений по этому поводу), и этот новый лидер инициирует новый раунд с новым баллотировочным числом.

В Paxos процедура принятия решения включает несколько раундов, что характерно для протоколов 2PC и 3PC, а также метод мажоритарного голосования из алгоритмов на основе кворума; он сводит их в единый протокол достижения консенсуса при наличии множества распределенных процессов. Отмечалось, что 2PC и 3PC (равно как и другие протоколы фиксации) – частные случаи Paxos. Далее мы опишем одно предложение для выполнения 2PC совместно с Paxos, позволяющее получить неблокирующий протокол 2PC, называемый Paxos 2PC.

В Paxos 2PC диспетчеры транзакций действуют как лидеры – по существу, это означает, что то, что мы раньше называли координатором, теперь именуется лидером. Основная особенность протокола – то, что лидер использует Paxos для достижения консенсуса и записи своего решения в реплицированный журнал. Первое важно, потому что для протокола необязательно, чтобы все участники были активны и принимали участие в принятии решения, – достаточно большинства, а остальные согласятся с принятым решением, когда восстановятся. Второе важно, потому что отказы лидера (координатора) больше не приводят к блокировке – если лидер выйдет из строя, то будет избран новый лидер, а состояние решения относительно транзакции будет доступно в реплицированном журнале на других узлах.

5.4.6. Архитектурные соображения

В предыдущих разделах мы обсуждали протоколы атомарной фиксации на абстрактном уровне. Теперь посмотрим, как эти протоколы можно реализовать в рамках нашей архитектурной модели. Это обсуждение будет включать определение интерфейса между алгоритмами управления конкурентностью и протоколами надежности. В этом смысле все обсуждаемое в этой главе связано с выполнением команд `Commit`, `Abort` и `recover`.

Точно определить выполнение этих команд непросто по двум причинам. Во-первых, для их корректной реализации необходимо рассматривать гораздо более детальную модель, чем та, что представлена нами. Во-вторых, общая схема реализации сильно зависит от процедур восстановления, которые реализует локальный диспетчер восстановления (ЛДВ). Например, реализация протокола 2PC поверх ЛДВ, в котором используется схема восстановления без задержки[1] и без сброса, полностью отличается от реализации поверх ЛДВ, в котором используется схема с задержкой и сбросом. Вариантов попросту слишком много. Поэтому в нашем обсуждении архитектуры мы ограни-

---

[1] Стратегии восстановления зависят от поведения диспетчера буферов. Если диспетчеру буферов не разрешено сохранять данные на диск до фиксации транзакции, то говорят о стратегии с задержкой (fix), иначе о стратегии без задержки (no-fix). Если диспетчер буферов обязан записать данные на диск в момент фиксации, то говорят о стратегии со сбросом (flush), а если он может сделать это позже, то о стратегии без сброса (no-flush). – *Прим. перев.*

чимся тремя вопросами: реализация концепций координатора и участника для протоколов фиксации и управления репликами в рамках архитектуры диспетчер транзакций – планировщик – локальный диспетчер восстановления; доступ координатора к журналу базы данных; изменения, которые необходимо внести в операции локального диспетчера восстановления.

Одна из возможных реализаций протоколов фиксации в нашей архитектурной модели – выполнять алгоритмы координатора и участника внутри диспетчеров транзакций в каждом узле. Тем самым будет обеспечена некоторая однородность выполнения операций распределенной фиксации. Однако при этом возникает лишнее взаимодействие между диспетчером транзакций участника и его планировщиком; объясняется это тем, что планировщику нужно решать, зафиксировать транзакцию или отменить. Поэтому предпочтительнее реализовать координатор как часть диспетчера транзакций, а участника – как часть планировщика. Если планировщик реализует строгий алгоритм управления конкурентностью (т. е. не допускает каскадной отмены), то он будет готов автоматически зафиксировать транзакцию при поступлении сообщения «prepare». Доказательство этого утверждения оставляем читателю в качестве упражнения. Однако даже этот вариант реализации координатора и участника вне процессора данных имеет свои проблемы. Первая – управление журналом базы данных. Напомним, что журнал базы данных обслуживается ЛДВ и диспетчером буферов. Однако описанная нами реализация протокола фиксации требует, чтобы диспетчер транзакций и планировщик тоже имели доступ к журналу. Одно из возможных решений этой проблемы – вести журнал фиксаций (который можно было бы назвать журналом распределенных транзакций), доступный диспетчеру транзакций и отделенный от журнала базы данных, который по-прежнему обслуживают ЛДВ и диспетчер буферов. Альтернатива – помещать записи протокола фиксации в тот же журнал базы данных. У этого варианта есть ряд преимуществ. Во-первых, ведется только один журнал, это упрощает алгоритмы, которые нужно реализовать для сохранения записей журнала в постоянном хранилище. Но еще важнее, что для восстановления после отказов в распределенной базе данных необходима кооперация локального диспетчера восстановления и планировщика (т. е. участника). Единый журнал базы данных может служить центральным репозиторием информации о восстановлении для обоих компонентов.

Вторая проблема, связанная с реализацией координатора внутри диспетчера транзакций, а участника как части планировщика, – интеграция с протоколами управления конкурентностью. Эта реализация основана на том, что именно планировщики решают, можно ли зафиксировать транзакцию. Это прекрасно для распределенных алгоритмов управления конкурентностью, когда в каждом узле имеется планировщик. Но в централизованных протоколах, например в централизованном 2PL, на всю систему приходится только один планировщик. В таком случае участников можно реализовать как часть процессоров данных (точнее, как часть локальных диспетчеров восстановления), что потребует модификации алгоритмов, реализованных ЛДВ, и, быть может, выполнения протокола 2PC. Детали оставляем в качестве упражнения.

Хранение записей протокола фиксации в журнале базы данных, обслуживаемом ЛДВ и диспетчером буферов, требует внесения некоторых изменений в алгоритмы ЛДВ. Это третья из рассматриваемых нами архитектурных проблем. Изменения зависят от типа алгоритма, используемого ЛДВ. В общем случае алгоритмы ЛДВ нужно модифицировать, так чтобы они раздельно обрабатывали команду «prepare» и решения global-commit (или global-abort). Кроме того, после восстановления ЛДВ должен читать журнал базы данных и информировать планировщик о состоянии каждой транзакции, чтобы можно было следовать изложенным выше процедурам восстановления. Рассмотрим эту функцию ЛДВ более пристально.

Первым делом ЛДВ должен определить, кто был размещен в отказавшем узле: координатор или участник. Эту информацию можно хранить в записи `Begin_transaction`. Затем ЛДВ должен найти последнюю запись, помещенную в журнал по время выполнения протокола фиксации. Если он не сможет найти даже запись `begin_commit` (в узле координатора) или запись `abort` либо `commit` (в узлах участников), то фиксация транзакции еще не начиналась. В таком случае ЛДВ может сам продолжить процедуру восстановления. Но если процесс фиксации начался, то восстановление следует препоручить координатору. Поэтому ЛДВ передает последнюю запись журнала планировщику.

5.5. Современные подходы к горизонтальному масштабированию управления транзакциями

Во всех представленных выше алгоритмах имеются узкие места на разных этапах обработки транзакций. Те, что реализуют концепцию сериализуемости, сильно ограничивают потенциал конкурентности из-за конфликтов между большими запросами, которые читают много данных, и транзакциями обновления. Например, аналитический запрос, выполняющий полный просмотр таблицы с предикатом, в котором не присутствует первичный ключ, приводит к конфликту с любым обновлением этой таблицы. Во всех алгоритмах необходим этап централизованной обработки, чтобы фиксировать транзакции одну за другой.

Это создает узкое место, потому что невозможно обрабатывать транзакции быстрее, чем это способен сделать один узел. Алгоритмы блокировки нуждаются в управлении взаимоблокировками, и во многих из них используется механизм обнаружения взаимоблокировок, который трудно реализовать в распределенной постановке, о чем мы подробно говорили. Алгоритм, представленный в последнем разделе, посвященном изоляции моментальных снимков, выполняет централизованную сертификацию, что опять-таки ведет к образованию узкого места.

Масштабирование выполнения транзакций для достижения высокой пропускной способности обработки транзакций в распределенной или параллельной системе давно уже вызывает интерес у исследователей. Относи-

тельно недавно стали появляться решения; в этом разделе мы обсудим два подхода: Google Spanner и LeanXcale. В обоих все свойства ACID реализованы масштабируемым и допускающим композицию способом. В обоих предложен новый метод сериализации транзакций, поддерживающий очень высокую пропускную способность (миллионы и даже миллиарды транзакций в секунду). В обоих подходах имеется способ снабжать фиксацию транзакций временной меткой и использовать эти метки для сериализации транзакций. В Spanner для пометки транзакций используется физическое время, а в LeanXcale – логическое. У физического времени есть то преимущество, что не требуется никакого взаимодействия, однако необходима высокая точность и очень надежная инфраструктура реального времени. Идея заключается в том, чтобы использовать физическое время в качестве временной метки и ждать некоторое время, определяемое точностью измерения, прежде чем сделать результат видимым транзакциям.

В LeanXcale транзакции помечаются логическим временем, а зафиксированные транзакции становятся видимы постепенно, по мере того как лакуны в порядке сериализации заполняются новыми зафиксированными транзакциями. Использование логического времени позволяет избежать зависимости от создания инфраструктуры реального времени.

5.5.1. Spanner

Spanner использует традиционную блокировку и протокол 2PC, а в качестве уровня изоляции предлагает сериализуемость. Поскольку блокировка приводит к высокой степени состязания между большими запросами чтения и транзакциями обновления, в Spanner также реализована многоверсионность. Чтобы избежать бутылочного горлышка централизованной сертификации, обновленным элементам данных после фиксации назначаются временные метки (с применением физического времени). Для этого в Spanner реализована внутренняя служба TrueTime, которая выдает текущее время и его текущую точность. Чтобы сделать службу TrueTime надежной и точной, используются атомные часы и GPS, поскольку для них характерны разные режимы отказа, так что они могут компенсировать друг друга. Например, у атомных часов имеется непрерывный уход, тогда как GPS теряет точность при определенных погодных условиях, когда ломается антенна и т. д. Текущее время, полученное от TrueTime, используется для формирования временных меток транзакций, которые вот-вот будут зафиксированы. Полученная точность нужна для компенсации в момент назначения временной метки: после получения локального времени необходимо подождать в течение периода неточности, обычно в районе 10 мс. Для борьбы с взаимоблокировками в Spanner применяется механизм избегания взаимоблокировок на основе подхода «завести и подождать»; это позволяет устранить узкое место, вызванное обнаружением взаимоблокировок.

Масштабируемость управления системой хранения в Spanner достигается благодаря использованию хранилища ключей и значений Google Bigtable (см. главу 11).

Многоверсионность реализована следующим образом. Частные версии элементов данных хранятся в каждом узле до момента фиксации. После фиксации начинает работать протокол 2PC, и в это время буферизованные операции записи распространяются всем участникам. Каждый участник ставит блокировки на обновленные элементы данных. После того как все блокировки захвачены, участник назначает временную метку фиксации, большую всех ранее назначенных. Координатор также захватывает блокировки записи. Захватив все блокировки и получив сообщения о готовности от всех участников, координатор выбирает временную метку, большую текущей плюс неточность и большую всех остальных локально назначенных меток. Координатор ждет, пока момент времени, соответствующий назначенной метке, останется в прошлом (напомним, что ожидание необходимо из-за неточности), а затем сообщает свое решение о фиксации клиенту.

С помощью многоверсионности Spanner также реализует транзакции чтения, которые читают снимок в текущий момент времени.

5.5.2. LeanXcale

В LeanXcale принят совершенно другой подход к масштабируемому управлению транзакциями. Во-первых, для пометки транзакций и задания видимости зафиксированных данных используется логическое время. Во-вторых, применяется изоляция моментальных снимков. В-третьих, все функции, интенсивно потребляющие ресурсы, например управление конкурентностью, ведение журнала, хранение и обработка запросов, являются полностью распределенными и параллельными и не нуждаются ни в какой координации.

Для получения логического времени в LeanXcale имеются две службы: задатчик последовательности фиксаций и сервер моментальных снимков. Задатчик распределяет временные метки фиксаций, а сервер снимков регулирует видимость зафиксированных данных, продвигая вперед снимок, видимый транзакциям.

Поскольку LeanXcale обеспечивает изоляцию снимков, ему необходимо обнаруживать конфликты типа запись-запись. Для этого реализован диспетчер конфликтов (который может быть распределенным). Каждый диспетчер конфликтов отвечает за проверку конфликтов для подмножества элементов данных. Идея в том, что диспетчер конфликтов получает от локальных диспетчеров транзакций (ЛДТ) просьбы проверить, конфликтует ли элемент данных, который планируется обновить, с текущими обновлениями. Если да, то ЛДТ отменяет транзакцию, иначе она может продолжать работу.

Функциональность системы хранения обеспечивается реляционным хранилищем ключей и значений KiVi. Таблицы горизонтально секционированы на секции, называемые *регионами*. Каждый регион хранится на одном сервере KiVi.

Фиксация управляется ЛДТ, который организует обработку следующим образом. Сначала ЛДТ получает временную метку фиксации из локального диапазона и использует ее для пометки множества записи транзакции, которое затем записывается в журнал. Для масштабирования ведения журнала при-

меняется несколько процессов-регистраторов. Каждый регистратор обслуживает подмножество ЛДТ. Регистратор отвечает ЛДТ, когда множество записи отправлено на долговечное хранение. LeanXcale реализует многоверсионность на уровне хранения. Как и в Spanner, в каждом узле хранятся частные копии. Когда множество записи становится постоянным, обновления, помеченные временными метками фиксации, распространяются на соответствующие серверы KiVi. После того как все обновления распространены, транзакция допускает чтение, если использована правильная временная метка начала (большая или равная текущей временной метке). Однако она все еще невидима. Затем ЛДТ уведомляет сервер снимков о том, что транзакция стала долговечной и допускающей чтение. Сервер снимков отслеживает текущий снимок, т. е. начальную временную метку, которая будет использоваться новыми транзакциями. Он также отслеживает долговечные и допускающие чтение транзакции, для которых временная метка фиксации больше текущего снимка. Если между текущим снимком и временной меткой нет лакун, то сервер снимков сдвигает снимок к этой временной метке. В этот момент зафиксированные данные, для которых временная метка фиксации меньше текущего снимка, становятся видны новым транзакциям, поскольку те получат временную метку начала в текущем снимке. Посмотрим, как это работает на примере.

Пример 5.5. Рассмотрим 5 транзакций, которые параллельно фиксируются с временными метками фиксации от 11 до 15. Пусть текущий снимок на сервере снимков равен 10. Локальные диспетчеры транзакций сообщают серверу снимков о долговечных и допускающих чтение транзакциях в порядке 11, 15, 14, 12, 13. Когда сервер снимков получает уведомление о транзакции с временной меткой фиксации 11, он продвигает снимок с 10 до 11, поскольку лакун нет. Получив транзакцию с временной меткой фиксации 15, он не может продвинуть снимок, поскольку тогда новые транзакции увидели бы несогласованное состояние, в котором отсутствуют изменения с метками от 12 до 14. Далее сервер узнает, что транзакция с меткой 14 стала долговечной и допускающей чтение. Но снимок все равно нельзя продвинуть. Следующей долговечной становится транзакция с меткой 13, и снимок продвигается до 15, потому что в порядке сериализации не осталось лакун. ♦

Заметим, что хотя этот алгоритм обеспечивает изоляцию снимков, он не гарантирует согласованность на уровне сеансов – желательное свойство, позволяющее транзакциям читать данные, записанные в том же сеансе ранее зафиксированными транзакциями. Для обеспечения сеансовой согласованности добавлен новый механизм. Когда в сеансе фиксируется транзакция, которая произвела обновления, сеанс ждет, пока снимок сдвинется дальше временной метки фиксации транзакции обновления, а затем начинает работу со снимка, гарантирующего, что будут видны сделанные в этом сеансе обновления.

5.6. Заключение

В этой главе мы обсудили вопросы, связанные с распределенной обработкой транзакций. Транзакция – это атомарная единица выполнения, которая

переводит базу данных из одного согласованного состояния в другое, тоже согласованное. Свойства ACID транзакций показывают, какие требования предъявляются к управлению ими. Понятие согласованности нуждается в определении поддержания целостности (мы дали его в главе 3), а также в алгоритмах управления конкурентностью. С управлением конкурентностью тесно связан вопрос об изоляции. Механизм распределенного управления конкурентностью в распределенных СУБД гарантирует поддержание согласованности распределенной базы данных и потому является одним из столпов распределенных СУБД. Мы рассмотрели распределенные алгоритмы управления конкурентностью трех типов: на основе блокировок, на основе упорядочения временных меток и оптимистические. Мы отметили, что алгоритмы на основе блокировок могут приводить к распределенным (или глобальным) взаимоблокировкам, и познакомились со способом обнаружения и разрешения взаимоблокировок.

Рассмотрение свойств долговечности и атомарности транзакций требует обсуждения надежности распределенных СУБД. Точнее, долговечность поддерживается различными протоколами фиксации, тогда как атомарность нуждается в разработке соответствующих протоколов восстановления. Мы познакомились с двумя протоколами фиксации, 2PC и 3PC, которые гарантируют атомарность и долговечность распределенных транзакций даже при наличии отказов. Один из этих алгоритмов (3PC) можно сделать неблокирующим, это позволило бы каждому узлу продолжать работу, не ожидая восстановления отказавшего узла. Интересный вопрос – производительность распределенных протоколов фиксации в плане накладных расходов на алгоритмы управления конкурентностью.

Вопрос о том, как добиться очень высокой пропускной способности выполнения транзакций в распределенных и параллельных СУБД, давно вызывал повышенный интерес, а в последние годы наметился некоторый прогресс. Мы обсудили два подхода к достижению этой цели, реализованных в системах Spanner и LeanXcale.

Несколько вопросов остались не рассмотренными в этой главе.

1. *Более сложные транзакционные модели*. Транзакционная модель, описанная в данной главе, обычно называется моделью *плоских транзакций*, поскольку транзакция имеет одну точку начала (`Begin_transaction`) и одну точку завершения (`End_transaction`). Большая часть работы по управлению транзакциями в базах данных сосредоточена вокруг плоских транзакций. Но существуют и более сложные транзакционные модели. Одна из них называется моделью *вложенных транзакций* – когда транзакция может включать другие транзакции с собственными точками начала и конца. Вложенные транзакции часто называют *подтранзакциями*. Вложенные транзакции образуют дерево, в котором самая внешняя транзакция является корнем, а подтранзакции представлены остальными узлами. Они различаются характеристиками завершения. Для одной категории – *замкнутых вложенных транзакций* – фиксация производится в восходящем порядке, через корень. Таким образом, подтранзакция начинается *после* своего родителя и заканчивается *до* него, а фиксация подтранзакций обусловлена фиксацией родителя. Семантика таких транзакций гарантирует атомарность на самом верх-

нем уровне. Альтернатива – открытая вложенность, которая ослабляет ограничение на атомарность верхнего уровня, действующее для замкнутых вложенных транзакций. Следовательно, открытая вложенная транзакция позволяет видеть частичные результаты вне самой транзакции. Примерами открытой вложенности являются саги и расщепленные транзакции.

Еще более сложно устроенные транзакции называются *потоками работ*. Термин «поток работ» (workflow), к сожалению, не имеет четкого и общепризнанного значения. В качестве рабочего определения можно считать, что это частично упорядоченное множество задач, которые коллективно работают над некоторым сложным процессом.

Хотя управление этими сложными транзакционными моделями – важный вопрос, он выходит за рамки книги, поэтому мы не станем рассматривать его в этой главе.

2. *Предположения относительно транзакций*. В наших обсуждениях мы не делали разницы между транзакциями чтения и транзакциями обновления. Есть возможность значительно повысить производительность транзакций, которые только читают данные, или систем с преобладанием таких транзакций над транзакциями обновления. Но эти вопросы выходят за рамки книги.

   Мы также рассматривали блокировки чтения и записи одинаково. Но их можно трактовать по-разному и разработать алгоритмы управления конкурентностью, допускающие «преобразование блокировок», когда транзакция сначала получает блокировку в одном режиме, а затем модифицирует ее в связи с изменением требований. Обычно блокировки чтения преобразуются в блокировки записи.
3. *Модели выполнения транзакций*. Во всех описанных нами алгоритмах предполагается вычислительная модель, в которой диспетчер транзакций в инициировавшем транзакцию узле координирует выполнение всех операций с базой данных, осуществляемых в рамках этой транзакции. Это называется *централизованным выполнением*. Но можно также рассмотреть модель *распределенного выполнения*, когда транзакция разлагается на множество подтранзакций, каждой из которых назначается узел, в котором диспетчер транзакции координирует ее выполнение. Интуитивно такая модель кажется более привлекательной, потому что допускает балансировку нагрузки на несколько узлов распределенной базы данных. Однако исследования производительности показывают, что распределенные вычисления дают выигрыш только при слабой нагрузке.
4. *Типы ошибок*. Мы рассматривали только отказы, обусловленные ошибками. Иными словами, предполагалось, что при проектировании и реализации систем (аппаратных и программных) приложены все возможные усилия, но из-за различных дефектов компонентов, изъянов проекта или операционной среды они работают неправильно. Такие отказы называются *невольными*. Но существуют также *намеренные отказы*, когда при проектировании или реализации не ставилась задача, чтобы она работала правильно. Например, при выполнении протокола

2PC участник, получивший от координатора сообщение, считает, что оно корректно: координатор функционирует и посылает участнику правильное сообщение, которое тот должен обработать. Единственная ошибка, о которой должен помнить участник, – отказ координатора или потеря сообщения от него. Это невольные отказы. Но если участник не может доверять полученным сообщениям, то он вынужден иметь дело с намеренными ошибками. Например, сайт участника может притвориться координатором и посылать вредоносные сообщения. Мы не обсуждали, какие меры нужно принять для борьбы с такими отказами. Методы обработки намеренных ошибок обычно называются *византийским соглашением*.

Помимо этих вопросов, имеется большой массив недавних работ по управлению транзакциями в различных средах (например, многоядерных системах в основной памяти). Мы не обсуждали их в этой главе, посвященной фундаментальным основам, но привели несколько ссылок в библиографических замечаниях.

5.7. Библиографические замечания

Управление транзакциями активно изучается с тех пор, как СУБД стали темой глубоких исследований. Имеется две великолепные книги на эту тему: [Gray and Reuter 1993] и [Weikum and Vossen 2001]. Классические учебники, посвященные этому предмету, – [Hadzilacos 1988] и [Bernstein et al. 1987]. Прекрасным дополнением к ним может служить книга [Bernstein and Newcomer 1997], в которой представлено углубленное обсуждение принципов обработки транзакций. Здесь же вы найдете более общий взгляд на обработку транзакций и мониторы транзакций, чем приведенный в этой книге и ориентированный исключительно на базы данных. Очень важной работой являются заметки по операционным системам баз данных [Gray 1979]. Среди прочего в них имеется ценная информация по управлению транзакциями.

Распределенное управление конкурентностью подробно рассмотрено в книге [Bernstein and Goodman 1981]. Она уже не переиздается, но ее текст имеется в интернете. Вопросы, рассмотренные в этой главе, гораздо более детально освещаются в работах [Cellary et al. 1988, Bernstein et al. 1987, Papadimitriou 1986] и [Gray and Reuter 1993].

Что касается описанных нами фундаментальных методов, то централизованный алгоритм 2PL впервые был предложен в работе [Alsberg and Day 1976], иерархическое обнаружение взаимоблокировок обсуждается в работе [Menasce and Muntz 1979], а обнаружение распределенных взаимоблокировок – в работе [Obermack 1982]. В обсуждении консервативного алгоритма упорядочения временных меток (УВМ) мы следовали работе [Herman and Verjus 1979]. Оригинальный многоверсионный алгоритм УВМ был предложен в работе [Reed 1978] и затем дополнительно формализован в работе [Bernstein and Goodman 1983]. В работе [Lomet et al. 2012] обсуждается, как реализовать многоверсионность поверх уровня конкурентности, на котором

реализован протокол 2PL, а в работе [Faleiro and Abadi 2015] то же самое делается поверх слоя, реализующего УВМ. Существуют также подходы, позволяющие реализовать версионность как общую инфраструктуру поверх любого метода управления конкурентностью [Agrawal and Sengupta 1993]. В работе [Bernstein et al. 1987] обсуждается, как реализовать оптимистические алгоритмы управления конкурентностью на базе блокировок, в работах [Thomas 1979] и [Kung and Robinson 1981] – реализации на основе временных меток. Обсуждение в разделе 5.2.4 следует работе [Ceri and Owicki 1982]. Оригинальное предложение, в котором описана изоляция моментальных снимков, – работа [Berenson et al. 1995]. В своем обсуждении алгоритма изоляции снимков мы следовали работе [Chairunnanda et al. 2014]. В работе [Binnig et al. 2014] обсуждается оптимизация, которую мы упомянули в конце этого раздела. Протоколы предполагаемой отмены и предполагаемой фиксации предложены в работах [Mohan and Lindsay 1983] и [Mohan et al. 1986]. Отказы узлов и восстановление после отказов – тема работ [Skeen and Stonebraker 1983] и [Skeen 1981], причем в последней предложен алгоритм 3PC вместе с его анализом. Обсуждаемые нами протоколы выборов координатора взяты из работы [Hammer and Shipman 1980] и [Garcia-Molina 1982]. Один из ранних обзоров согласованности в условиях разделения сети – работа [Davidson et al. 1985]. В работе [Thomas 1979] предложен оригинальный метод мажоритарного голосования, а рассмотренный нами вариант протокола в нереплицированной базе данных предложен в работе [Skeen 1982a]. Идея распределенного журнала транзакций в разделе 5.4.6 заимствована из работ [Bernstein et al. 1987] и [Lampson and Sturgis 1976].

Обсуждение управления транзакциями в системе R* взято из работы [Mohan et al. 1986]; система NonStop SQL представлена в работах [Tandem 1987, 1988, Borr 1988]. В статье [Bernstein et al. 1980b] подробно обсуждается система SDD-1. Более современные системы Spanner и LeanXcale (см. раздел 5.5) описаны соответственно в работах [Corbett et al. 2013] и [Jimenez-Peris and Patiño Martinez 2011].

Более сложные транзакционные модели и различные примеры представлены в книге [Elmagarmid 1992]. Вложенные транзакции рассмотрены также в работе [Lynch et al. 1993]. Замкнутые вложенные транзакции изучались в работе [Moss 1985], модель открытых вложенных транзакций – саг – предложена в работах [Garcia-Molina and Salem 1987], [Garcia-Molina et al. 1990], а расщепленные транзакции – в работе [Pu 1988]. Модели вложенных транзакций и соответствующие алгоритмы управления конкурентностью стали предметом изучения в нескольких работах. Конкретные результаты можно найти в работах [Moss 1985, Lynch 1983b, Lynch and Merritt 1986, Fekete et al. 1987a,b, Goldman 1987, Beeri et al. 1989, Fekete et al. 1989] и [Lynch et al. 1993]. Хорошее введение в системы рабочих потоков – статья [Georgakopoulos et al. 1995], эта тема рассмотрена также в работах [Dogac et al. 1998] и [van Hee 2002].

Теме управления транзакциями с семантическими знаниями посвящены работы [Lynch 1983a, Garcia-Molina 1983] и [Farrag and Özsu 1989]. Обработка транзакций чтения обсуждается в работах [Garcia-Molina andWiederhold 1982]. В группах транзакций [Skarra et al. 1986, Skarra 1989] также использу-

ется критерий корректности – *семантические паттерны* – более слабый, чем сериализуемость. Этот же класс алгоритмов применяется в системе ARIES [Haderle et al. 1992]. В частности, в работе [Rothermel and Mohan 1989] обсуждается ARIES в контексте вложенных транзакций. Из других «ослабленных» критериев корректности упомянем эпсилон-сериализуемость [Ramamritham and Pu 1995, Wu et al. 1997] и модель NT/PV [Kshemkalyani and Singhal 1994]. Алгоритм, основанный на упорядочении транзакций с помощью чисел сериализации, обсуждается в работе [Halici and Dogac 1989].

В двух книгах обсуждается производительность механизмов управления конкурентностью с упором на централизованные системы: [Kumar 1996, Thomasian 1996]. В работе [Kumar 1996] основной акцент сделан на централизованные СУБД; производительность методов распределенного управления конкурентностью обсуждается в работах [Thomasian 1996] и [Cellary et al. 1988]. Ранний, но достаточно полный обзор управления взаимоблокировками приведен в работе [Isloor and Marsland 1980]. Большинство работ по распределенному управлению взаимоблокировками посвящены их обнаружению и разрешению (см., например, [Obermack 1982, Elmagarmid et al. 1988]). Обзоры наиболее важных алгоритмов включены в работы [Elmagarmid 1986], [Knapp 1987] и [Singhal 1989].

Тема изоляции моментальных снимков (SI) привлекала много внимания в последние годы. Хотя Oracle реализовал SI еще в ранних версиях, сама концепция была формально определена только в работе [Berenson et al. 1995]. В этой главе не нашло отражения одно направление работ – как добиться сериализуемого выполнения, когда в качестве критерия корректности используется SI. С этой целью алгоритм управления конкурентностью модифицируется за счет обнаружения и предотвращения аномалий, вызываемых SI и способных привести к несогласованности данных [Cahill et al. 2009, Alomari et al. 2009, Revilak et al. 2011, Alomari et al. 2008]. Подобные методы начали встраиваться в реальные системы, например PostgreSQL [Ports and Grittner 2012]. Первый алгоритм управления конкурентностью для SI, опубликованный в работе [Schenkel et al. 2000], был ориентирован на управление конкурентностью в системах интеграции данных, использующих SI. Наше обсуждение основано на системе ConfluxDB [Chairunnanda et al. 2014]; другая работа в этом направлении – [Binnig et al. 2014], где разработаны более тонкие методы.

В работе [Kohler 1981] имеется общее обсуждение вопросов надежности в распределенных системах баз данных. В работе [Hadzilacos 1988] предложена формализация понятия надежности. Связанные с надежностью аспекты системы R* описаны в работе [Traiger et al. 1982], а в работе [Hammer and Shipman 1980] то же самое сделано для системы SDD-1.

Более подробные сведения о функциях локального диспетчера восстановления можно найти в работах [Verhofstadt 1978, Härder and Reuter 1983]. Реализация локальных функций восстановления в System R описана в работе [Gray et al. 1981].

Протокол двухфазной фиксации впервые описан в работе [Gray 1979]. Его модификации представлены в работах [Mohan and Lindsay 1983]. Определение трехфазной фиксации дано в работах [Skeen 1981, 1982b]. Формальные

результаты о существовании неблокирующих протоколов завершения приведены в работе [Skeen and Stonebraker 1983]. Протокол Paxos первоначально предложен в работе [Lamport 1998]. Эта статья считается трудной для чтения, что привело к появлению ряда других статей с описанием протокола. В работе [Lamport 2001] содержится значительно упрощенное описание, а в работе [Van Renesse and Altinbuken 2015] – описание, занимающее среднюю позицию между двумя крайностями, именно ее лучше всего изучить. Протокол Paxos 2PC, который мы описали очень кратко, предложен в работе [Gray and Lamport 2006]. Рекомендуемое обсуждение вопроса о конструировании системы на основе Paxos см. в работах [Chandra et al. 2007] и [Kirsch and Amir 2008]. Существует много версий Paxos – слишком много, чтобы перечислять их все, – поэтому Paxos называют «семейством протоколов». Мы не станем приводить ссылки на все версии. Заметим также, что Paxos – не единственный алгоритм достижения консенсуса; имеется целый ряд альтернатив, особенно популярны стали алгоритмы на основе блокчейна (см. обсуждение блокчейна в главе 9). Этот список быстро растет, поэтому мы не приводим ссылок. Один алгоритм, Raft, был предложен как ответ на сложность и предполагаемую трудность понимания Paxos. Оригинальный вариант описан в статье [Ongaro and Ousterhout 2014] и изящно изложен в главе 23 книги [Silberschatz et al. 2019].

Как уже было сказано, мы не обсуждаем в этой главе византийские ошибки. В протоколе Paxos такие ошибки тоже не рассматриваются. Хорошее описание того, как с ними обращаться, имеется в работе [Castro and Liskov 1999].

Что касается более свежих работ на тему этой главы, упомянем статью [Tu et al. 2013], где обсуждается вертикально масштабируемая обработка транзакций на одной многоядерной машине. В работе [Kemper and Neumann 2011] обсуждаются вопросы управления транзакциями в гибридной OLAP/OLTP среде в контексте системы HyPer, работающей в основной памяти. В работе [Larson et al. 2011] тот же вопрос обсуждается в контексте системы Hekaton. В системе E-store, которую мы обсуждали в главе 2 в связи с адаптивным секционированием данных, также решается задача управления транзакциями в секционированной распределенной СУБД. Как мы уже отмечали, в E-store используется подсистема Squall [Elmore et al. 2015], которая учитывает транзакции при решении вопроса о перемещении данных. В работе [Thomson and Abadi 2010] предложена система Calvin, в которой метод избегания взаимоблокировок сочетается с алгоритмами управления конкурентностью, чтобы получить истории, которые гарантированно эквивалентны предопределенному последовательному упорядочению в реплицированной распределенной СУБД.

Упражнения

Задача 5.1. Какие из следующих историй эквивалентны по конфликтам:

$H_1 = \{W_2(x), W_1(x), R_3(x), R_1(x), W_2(y), R_3(y), R_3(z), R_2(x)\}$;

$H_2 = \{R_3(z), R_3(y), W_2(y), R_2(z), W_1(x), R_3(x), W_2(x), R_1(x)\}$;
$H_3 = \{R_3(z), W_2(x), W_2(y), R_1(x), R_3(x), R_2(z), R_3(y), W_1(x)\}$;
$H_4 = \{R_2(z), W_2(x), W_2(y), W_1(x), R_1(x), R_3(x), R_3(z), R_3(y)\}$?

Задача 5.2. Какие из историй H_1–H_4 сериализуемы?

Задача 5.3. Приведите пример истории двух полных транзакций, которая не разрешена строгим планировщиком 2PL, но допускается базовым планировщиком 2PL.

Задача 5.4 (*). Говорят, что история *H допускает восстановление*, если всякий раз, как транзакция T_i читает (некоторый элемент x) из транзакции T_j ($i \neq j$) в H и C_i встречается в H, то $C_j \prec_S C_i$. T_i «читает x из» T_j в H, если

1) $W_j(x) \prec_H R_i(x)$ и
2) A_j не $\prec_H R_i(x)$ и
3) если существует $W_k(x)$ такое, что $W_j(x) \prec_H W_k(x) \prec_H R_i(x)$, то $A_k \prec_H R_i(x)$.

Какие из следующих историй допускают восстановление:

$H_1 = \{W_2(x), W_1(x), R_3(x), R_1(x), C_1, W_2(y), R_3(y), R_3(z), C_3, R_2(x), C_2\}$;
$H_2 = \{R_3(z), R_3(y), W_2(y), R_2(z), W_1(x), R_3(x), W_2(x), R_1(x), C_1, C_2, C_3\}$;
$H_3 = \{R_3(z), W_2(x), W_2(y), R_1(x), R_3(x), R_2(z), R_3(y), C_3, W_1(x), C_2, C_1\}$;
$H_4 = \{R_2(z), W_2(x), W_2(y), C2, W_1(x), R_1(x), A_1, R_3(x), R_3(z), R_3(y), C_3\}$?

Задача 5.5 (*). Спроектируйте алгоритмы диспетчеров транзакций и диспетчеров блокировки для распределенного подхода к двухфазной блокировке.

Задача 5.6 (**). Модифицируйте централизованный алгоритм 2PL, так чтобы он обрабатывал фантомное чтение. Фантомное чтение имеет место, когда внутри транзакции выполняются две операции чтения и результат второй операции содержит кортежи, отсутствующие в результате первой. Рассмотрим следующий пример, который может возникнуть в системе бронирования авиабилетов. Во время выполнения транзакции T_1 в таблице FC производится поиск имен пассажиров, заказавших специальное питание. Возвращается множество значений атрибута CNAME пассажиров, удовлетворяющих критерию поиска. Пока T_1 выполняется, транзакция T_2 вставляет в FC новые кортежи, соответствующие пассажирам, заказавшим специальное питание, и фиксируется. Если впоследствии T_1 снова выполнит тот же запрос, она получит множество значений CNAME, отличающееся от первоначального. Таким образом, в базе данных появились «фантомные» кортежи.

Задача 5.7. Для правильной работы алгоритмов управления конкурентностью на основе упорядочения временных меток требуется либо наличие точных часов в каждом узле, либо наличие глобальных часов, к которым могут обращаться все узлы (часы могут быть и счетчиком). Предположим, что в каждом узле есть свои часы, которые «тикают» с интервалом 0.1 с. Если все локальные часы повторно синхронизируются раз в 24 часа, то при каком максимальном суточном уходе часов (в секундах) гарантируется правильная синхронизация транзакций механизмом на основе временных меток?

Задача 5.8 (**). Включите стратегию обнаружения распределенной блокировки, описанную в этой главе, в распределенные алгоритмы 2PL, которые

вы спроектировали в задаче 5.5.

Задача 5.9. Объясните связь между требованием диспетчера транзакций к системе хранения и размером транзакции (количеством операций в транзакции), если диспетчер транзакций пользуется оптимистическим упорядочением временных меток для управления конкурентностью.

Задача 5.10 (*). Спроектируйте алгоритмы планировщика и диспетчера транзакций для оптимистического распределенного управления конкурентностью, описанного в этой главе.

Задача 5.11. Напомним (см. раздел 5.6), что во всех описаниях в этой главе предполагалась централизованная модель вычислений. Как следовало бы изменить распределенные алгоритмы 2PL-диспетчера транзакций и диспетчера блокировок, если бы использовалась распределенная модель выполнения?

Задача 5.12. Иногда можно услышать утверждение, что сериализуемость – чрезмерно ограничительный критерий корректности. Можете ли вы привести примеры распределенных историй, которые были бы корректны (т. е. поддерживают как согласованность локальных баз данных, так и их взаимную согласованность), но не сериализуемы?

Задача 5.13 (*). Обсудите протокол завершения в случае отказа узла для 2PC на примере распределенной топологии связи.

Задача 5.14 (*). Спроектируйте протокол 3PC, используя линейную топологию связи.

Задача 5.15 (*). В нашем изложении централизованного протокола завершения для 3PC на первом шаге состояние координатора передавалось всем участникам. Участники переходят в новое состояние, соответствующее состоянию координатора. Можно спроектировать протокол завершения, так что координатор будет не посылать информацию о своем состоянии участникам, а запрашивать у участников их состояние. Модифицируйте протокол завершения в этом ключе.

Задача 5.16 (**). В разделе 5.4.6 мы утверждали, что планировщик, который реализует строгий алгоритм управления конкурентностью, всегда готов зафиксировать транзакцию, получив от координатора сообщение «prepare». Докажите это утверждение.

Задача 5.17 (**). Считая, что координатор реализован как часть диспетчера транзакций, а участник – как часть планировщика, спроектируйте алгоритмы диспетчера транзакций, планировщика и локального диспетчера восстановления для нереплицированной распределенной СУБД при следующих предположениях:

- **a)** планировщик реализует распределенный алгоритм управления конкурентностью на основе (строгой) двухфазной блокировки;
- **b)** записи протокола фиксации помещают в центральный журнал базы данных с помощью ЛДВ, когда тот вызывается планировщиком;

c) ЛДВ может реализовать любой из упомянутых протоколов (например, с задержкой без сброса или еще какой-то). Однако он модифицирован, так чтобы поддерживались процедуры распределенного восстановления, как описано в разделе 5.4.6.

Задача 5.18 (*). Напишите детальные алгоритмы для локального диспетчера восстановления без фиксирования и без сброса.

Задача 5.19 (**). Предположим, что:

a) планировщик реализует централизованное управление конкурентностью на основе двухфазной блокировки;

b) ЛДВ реализует протокол без задержки и без сброса.

Напишите детальные алгоритмы диспетчера транзакции, планировщика и локального диспетчера восстановления.

Глава 6

Репликация данных

В предыдущих главах мы говорили, что распределенные базы данных обычно реплицируются. Репликация преследует несколько целей.

1. **Доступность системы.** В главе 1 отмечалось, что распределенная СУБД может исключить точки общего отказа с помощью репликации данных, так чтобы копии элементов данных были доступны в нескольких узлах. Тогда, даже если некоторые узлы выйдут из строя, данные можно будет получить от других узлов.
2. **Производительность.** Выше мы видели, что одна из основных составляющих времени ответа – затраты на передачу данных. Репликация позволяет разместить данные ближе к точкам доступа, а значит, локализовать большинство операций доступа, уменьшив тем самым время ответа.
3. **Масштабируемость.** Если система расширяется географически и количество узлов растет (а значит, растет и количество запросов), то репликация позволяет справиться с ростом, удержав время ответа в разумных пределах.
4. **Требования со стороны приложений.** Наконец, необходимость репликации может быть продиктована приложениями, спецификация которых требует наличия нескольких копий данных.

Хотя преимущества репликации очевидны, синхронизация копий ставит трудные проблемы. Чуть ниже мы их обсудим, но сначала рассмотрим модель выполнения в реплицированных базах данных. У каждого реплицированного элемента данных x есть несколько копий x_1, x_2, …, x_n. Будем называть x *логическим элементом данных*, а его копии (или реплики)[1] – *физическими элементами данных*. Если необходимо обеспечить прозрачность репликации, то пользовательские транзакции выполняют операции чтения и записи над логическим элементом данных x. Протокол управления репликами отвечает за отображение этих операций на операции чтения и записи физических элементов данных x_1, x_2, …, x_n. Таким образом, система ведет себя так, будто имеется единственная копия каждого элемента данных – это называется *единым образом системы*, или *эквивалентностью всех копий* (one-copy equivalence). Конкретные реализации интерфейсов `Read` и `Write` монитора

[1] В этой главе слова «копия», «реплика» и «физический элемент данных» употребляются как синонимы.

транзакций зависят от протокола репликации, и мы обсудим эти различия в соответствующих разделах.

Существует целый ряд решений и факторов, влияющих на проектирование протоколов репликации. Некоторые из них мы обсуждали в предыдущих главах, другие обсудим здесь.

- **Структура базы данных.** В главе 2 мы говорили, что распределенная база данных может быть полностью или частично реплицированной. В случае частично реплицированной базы данных количество физических элементов данных для каждого логического может меняться, а некоторые данные могут даже не реплицироваться. В этом случае транзакции, обращающиеся только к нереплицированным элементам данных, являются *локальными* (поскольку они могут выполняться локально в одном узле), и их выполнение нас здесь интересовать не будет. Транзакции, обращающиеся к реплицированным элементам данных, должны выполняться в нескольких узлах, это *глобальные транзакции*.
- **Согласованность базы данных.** Когда глобальная транзакция обновляет копии элемента данных в разных узлах, значения этих копий в один и тот же момент времени могут различаться. Говорят, что реплицированная база данных *взаимно согласована*, если все реплики каждого элемента данных имеют в точности одинаковые значения. Критерии взаимной согласованности различаются точностью синхронизации реплик. Одни гарантируют взаимную согласованность реплик в момент фиксации транзакции обновления – это называется *сильной согласованностью*. В других требования не такие строгие, они называются критериями *слабой согласованности*.
- **Когда выполняются обновления.** Фундаментальное решение при проектировании протокола репликации – где сначала выполняются обновления. Подход называется *централизованным*, если обновление сначала выполняется в *главной* копии, и *распределенным*, если разрешено обновлять любую реплику. Централизованные методы подразделяются на методы с *одной главной копией*, когда в системе существует единственная главная копия базы данных, и методы с *ведущей копией*, когда главные копии разных элементов данных могут находиться в разных местах[1].
- **Распространение обновлений.** После того как обновления произведены в какой-то реплике (главной или иной), нужно распространить их на все остальные. Тут есть два варианта: *энергичный* (eager) и *ленивый* (lazy). При энергичном распространении все обновления производятся в контексте глобальной транзакции, инициировавшей операцию записи. Поэтому если транзакция зафиксирована, то обновления при-

[1] В литературе централизованные методы называются методами с *одной главной копией* (single master), а распределенные – методами с *несколькими главными копиями* (multimaster), или *с обновлением всюду* (update anywhere). Эти термины, особенно «с одной главной копией», вводят в заблуждение, потому что относятся к различным архитектурам реализации централизованных протоколов (подробнее см. раздел 6.2.3). Поэтому мы предпочитаем термины «централизованный» и «распределенный».

менены ко всем копиям. С другой стороны, при ленивом распространении обновления производятся в какой-то момент после фиксации транзакции. Энергичные методы далее подразделяются в зависимости от того, когда операции записи передаются на другие реплики – иногда это делается отдельно для каждой операции, а иногда пакетом, когда все операции записи распространяются из точки фиксации.

- **Степень прозрачности репликации.** В некоторых протоколах репликации требуется, чтобы каждое пользовательское приложение знало о главном узле, которому передаются операции транзакции. Такие протоколы обеспечивают только *ограниченную прозрачность репликации*. Другие протоколы обеспечивают *полную прозрачность репликации*, задействуя диспетчер транзакций в каждом узле. В этом случае пользовательские приложения отправляют транзакции своим локальным ДТ, а не главному узлу.

В разделе 6.1 мы обсудим вопросы согласованности в реплицированных базах данных, а в разделе 6.2 сравним централизованное и распределенное обновления, а также различные варианты распространения обновлений. Это подведет нас к обсуждению конкретных протоколов в разделе 6.3. В разделе 6.4 мы рассмотрим использование примитивов групповой коммуникации для уменьшения накладных расходов на передачу сообщений в протоколах репликации. В этих разделах предполагается, что никаких ошибок нет, поэтому можно сосредоточиться на самих протоколах репликации. В разделе 6.5 мы поговорим об ошибках и посмотрим, как нужно модифицировать протоколы для их обработки.

6.1. Согласованность реплицированных баз данных

При обсуждении согласованности реплицированных баз данных возникает два вопроса. Один – взаимная согласованность, т. е. сходимость значений физических элементов данных, соответствующих одному логическому. Второй – согласованность транзакций, обсуждавшаяся в главе 5. Сериализуемость, которая была введена как критерий согласованности транзакций, придется пересмотреть в контексте реплицированных баз данных. Кроме того, существуют связи между взаимной согласованностью и согласованностью транзакций. В этом разделе мы сначала обсудим подходы к взаимной согласованности, а затем займемся переопределением согласованности транзакций и ее связями со взаимной согласованностью.

6.1.1. Взаимная согласованность

Выше мы уже отмечали, что критерий взаимной согласованности для реплицированных баз данных может быть сильным или слабым. Каждый из них

полезен, но только в разных классах приложений с разными требованиями к согласованности.

Критерий сильной взаимной согласованности требует, чтобы все копии элемента данных имели одно и то же значение в конце выполнения транзакции обновления. Этого можно достичь разными средствами, но чаще всего используется протокол 2PC в точке фиксации транзакции обновления.

Критерий слабой взаимной согласованности не требует, чтобы значения всех реплик данных были одинаковы сразу после завершения транзакции. Требуется другое – если операции обновления на некоторое время прекращаются, то *в конечном счете* значения данных должны стать одинаковыми. Это обычно называют *согласованностью в конечном счете* – реплики могут отличаться на протяжении некоторого времени, но рано или поздно сходятся к одному значению. Определить эту концепцию формально довольно трудно, но следующее определение Саито и Шапиро, пожалуй, является настолько точным, насколько это вообще возможно:

> Реплицированный [элемент данных] называется *согласованным в конечном счете*, если он удовлетворяет следующим условиям в предположении, что начальное состояние всех реплик было одинаково.
> - В любой момент для каждой реплики существует префикс [истории], эквивалентный префиксу [истории] любой другой реплики. Он называется *зафиксированным префиксом* реплики.
> - Зафиксированный префикс любой реплики монотонно возрастает со временем.
> - Все неотмененные операции в зафиксированном префиксе удовлетворяют своим предусловиям.
> - Для каждой совершенной операции α либо α, либо результат ее отмены в конечном счете будет включен в зафиксированный префикс.

Отметим, что это определение согласованности в конечном счете довольно сильное – особенно требования, чтобы префиксы истории были одинаковы в любой момент времени и чтобы зафиксированный префикс монотонно возрастал. Многие системы, заявляющие, что обеспечивают согласованность в конечном счете, эти требования нарушают.

Эпсилон-сериализуемость (ЭС) позволяет запросу видеть несогласованные данные, пока реплики обновляются, но требует, чтобы реплики сходились к состоянию, эквивалентному одной копии, когда обновления будут распространены во все копии. Ошибка при чтении значений ограничена значением эпсилон (ε), определенным в терминах количества обновлений (операций записи), которые запрос «пропустил». Пусть дана транзакция чтения (запрос) T_Q, обозначим T_U множество всех транзакций обновления, которые выполняются одновременно с T_Q. Если $RS(T_Q) \cap WS(T_U) \neq \emptyset$ (T_Q читает какие-то элементы данных, пока транзакция из T_U обновляет (возможно, другие) копии этих данных), то налицо конфликт чтение-запись, и T_Q может прочитать несогласованные данные. Несогласованность ограничена изменениями, произведенными T_U. Очевидно, что ЭС не жертвует согласованностью базы данных, но лишь разрешает транзакциям чтения читать несогласованные данные. Поэтому говорят, что ЭС не ослабляет согласованность базы данных, а «раздвигает» ее пределы.

Обсуждались и другие более слабые границы. Даже были предложения разрешить пользователям самим определять *ограничения актуальности*, подходящие конкретным приложениям, и включить в протоколы репликации средства их проверки. Возможны следующие типы ограничений актуальности:

- **временные ограничения.** Пользователи могут согласиться на расхождение значений физических копий, но не более чем на заданное время: x_i может отражать результат обновления в момент t, а x_j – в момент $t - \Delta$, и это допустимо;
- **ограничения на значения.** Допустимо, чтобы значения всех физических элементов данных различались не более чем на некоторую величину. Пользователь может считать базу данных взаимно согласованной, если значения расходятся не более чем на заданную абсолютную или относительную величину;
- **ограничения на расхождение нескольких элементов данных.** Для транзакций, читающих несколько элементов данных, может быть разрешено расхождение между временными метками любых двух элементов, не превышающее заданный порог (т. е. они должны быть обновлены в течение ограниченного промежутка времени), или – в случае вычисления агрегата – если вычисляемый по элементу данных агрегат не слишком сильно отклоняется от самого последнего значения (т. е. даже если отдельные физические копии рассинхронизированы на большую величину, это считается нормальным, коль скоро вычисление конкретного агрегата не выходит за пределы заданного диапазона).

Важным критерием при анализе протоколов, допускающих расхождение реплик, является *степень актуальности*. Степень актуальности данной реплики x_i в момент t определяется как отношение числа обновлений, примененных к x_i в момент t, к общему числу обновлений.

6.1.2. Взаимная согласованность и согласованность транзакций

Взаимная согласованность, определяемая в этом разделе, и согласованность транзакций, рассмотренная в главе 5, связаны, но не тождественны. Под взаимной согласованностью понимается сходимость реплик к одному значению, а для согласованности транзакций необходимо, чтобы глобальная история выполнения была сериализуемой. Может быть так, что реплицированная СУБД гарантирует взаимную согласованность элементов данных в момент фиксации транзакции, но история выполнения при этом не является глобально сериализуемой. Это демонстрирует следующий пример.

Пример 6.1. Рассмотрим три узла (A, B, C) и три элемента данных (x, y, z), распределенных следующим образом: в узле A размещен x, в узле B – x и y, а в узле C – x, y, z. Будем использовать идентификаторы узлов как нижние индексы элементов данных, относящихся к соответствующей реплике.

Рассмотрим такие три транзакции:

T_1:	$x \leftarrow 20$	T_2:	Read(x)	T_3:	Read(x)
	Write(x)		$y \leftarrow x + y$		Read(y)
	Commit		Write(y)		$z \leftarrow (x * y)/100$
			Commit		Write(z)
					Commit

Заметим, что операцию Write из T_1 необходимо выполнить во всех трех узлах (поскольку x реплицирован на три узла), операцию Write из T_2 – в узлах B и C, а операцию Write из T_3 – только в C. Предполагается модель выполнения транзакций, в которой транзакции могут читать свои локальные реплики, но обновлять должны все реплики.

Предположим, что в узлах сгенерированы следующие три локальные истории:

$H_A = \{W_1(x_A), C_1\}$;
$H_B = \{W_1(x_B), C_1, R_2(x_B), W_2(y_B), C_2\}$;
$H_C = \{W_2(y_C), C_2, R_3(x_C), R_3(y_C), W_3(z_C), C_3, W_1(x_C), C_1\}$.

Порядок сериализации в H_B $T_1 \rightarrow T_2$, в H_C $T_2 \rightarrow T_3 \rightarrow T_1$. Поэтому глобальная история не сериализуема. Однако база данных взаимно согласована. Предположим, к примеру, что в начальный момент $x_A = x_B = x_C = 10$, $y_B = y_C = 15$, $z_C = 7$. При тех историях, что показаны выше, конечные значения будут равны $x_A = x_B = x_C = 20$, $y_B = y_C = 35$, $z_C = 3.5$. Все физические копии (реплики) действительно сошлись к одному и тому же значению. ♦

Конечно, может случиться, что база данных взаимно не согласована и история выполнения глобально не сериализуема, как показано в примере ниже.

Пример 6.2. Рассмотрим два узла (A и B) и один элемент данных (x), реплицированный на оба узла (x_A и x_B). И рассмотрим такие две транзакции:

T_1:	Read(x)	T_2:	Read(x)
	$x \leftarrow x + 5$		$x \leftarrow x * 10$
	Write(x)		Write(x)
	Commit		Commit

Предположим, что в двух узлах сгенерированы такие две локальные истории (используется та же модель выполнения, что в предыдущем примере):

$H_A = \{R_1(x_A), W_1(x_A), C_1, R_2(x_A), W_2(x_A), C_2\}$;
$H_B = \{R_2(x_B), W_2(x_B), C_2, R_1(x_B), W_1(x_B), C_1\}$.

Хотя обе истории последовательные, T_1 и T_2 в них сериализованы в противоположном порядке, поэтому глобальная история не сериализуема. Нарушается и взаимная согласованность. Допустим, что значение x до выполнения этих транзакций было равно 1. В конце выполнения x будет равно 60 в узле A и 15 в узле B. Таким образом, в этом примере глобальная история не сериализуема **и** базы данных взаимно не согласованы. ♦

С учетом этого наблюдения критерий согласованности транзакций, сформулированный в главе 5, обобщается на реплицированные базы данных под названием *сериализуемость как в одной копии* (one-copy serializability – 1SR). Это означает, что результат выполнения транзакций над реплицированными элементами данных должен быть таким, как если бы они выполнялись по одной над одним набором элементов данных. Иными словами, истории эквивалентны некоторому последовательному выполнению над нереплицированными элементами данных.

Изоляция моментальных снимков, введенная в главе 5, также обобщена на реплицированные базы данных и используется в качестве альтернативного критерия согласованности транзакций в контексте реплицированных баз данных. Определена и более слабая форма сериализуемости, названная (*RC-*)*сериализуемостью с ослабленной конкурентностью*. Она соответствует уровню изоляции «чтение зафиксированных данных» (Read Committed).

6.2. Стратегии управления обновлениями

Выше мы говорили, что протоколы репликации можно классифицировать в соответствии со способом распространения обновлений на реплики (ленивое или энергичное) и с тем, где разрешено производить обновления (централизованное или распределенное). Эти два решения обычно называют стратегиями *управления обновлениями*. В этом разделе мы обсудим альтернативы, а в следующем опишем протоколы.

6.2.1. Энергичное распространение обновлений

При энергичном распространении обновлений изменения применяются ко всем репликам в контексте транзакции обновления. Следовательно, после фиксации транзакции обновления все копии будут иметь одно и то же значение. Обычно в точке фиксации методы энергичного распространения пользуются протоколом 2PC, но, как мы увидим ниже, есть и другие способы достичь согласия. Кроме того, в энергичном режиме можно использовать *синхронное* распространение каждого обновления, применяя его ко всем репликам одновременно (в момент выполнения `Write`), или *отложенное* распространение, когда обновления применяются к одной реплике в момент выполнения, а к другим репликам все обновления применяются в пакетном режиме в конце транзакции. Отложенное распространение можно реализовать, включив обновления в сообщение «Prepare-to-Commit» в начале выполнения 2PC.

Энергичное распространение обычно гарантирует соблюдение критерия взаимной согласованности. Поскольку все реплики взаимно согласованы в конце транзакции обновления, впоследствии можно читать из любой копии (т. е. можно заменить $R(x)$ значением $R(x_i)$ для любого x_i). Однако $W(x)$ должна быть применена ко всем репликам (т. е. $W(x_i)$, $\forall x_i$). Поэтому протоколы, основанные на энергичном распространении обновлений, называются протоколами *чтение одной / запись всех* (read-one/write-all –ROWA). У энергич-

ного распространения обновлений есть три преимущества. Во-первых, они обычно гарантируют взаимную согласованность в соответствии с критерием 1SR, следовательно, отсутствуют несогласованности транзакций. Во-вторых, транзакция может прочитать локальную копию элемента данных (если таковая имеется), и это наверняка будет актуальное значение. Поэтому можно обойтись без удаленного чтения. Наконец, изменения вносятся в реплики атомарно, поэтому восстановление после отказов можно осуществлять по протоколам, рассмотренным в предыдущей главе.

Основной недостаток энергичного распространения обновлений заключается в том, что транзакция обязана обновить все копии, перед тем как завершится. Отсюда вытекает два следствия. Во-первых, увеличивается время ответа, потому что транзакция участвует в выполнении протокола 2PC и потому что скорость обновления ограничена самой медленной машиной. Во-вторых, если какая-то копия недоступна, то транзакция не может завершиться. В главе 5 мы говорили, что если имеется возможность различить отказы узлов и отказы сети, то можно завершать транзакцию, как только какая-нибудь одна реплика станет недоступной (напомним, что недоступность более одного узла делает протокол 2PC блокирующим), но, вообще говоря, различить эти два типа отказов невозможно.

6.2.2. Ленивое распространение обновлений

В случае ленивого распространения обновлений не все обновления реплик производятся в контексте транзакции обновления. Иными словами, транзакция не ждет, пока обновления будут применены ко всем копиям, а фиксируется, как только обновлена одна реплика. Распространение на другие реплики осуществляется *асинхронно* по отношению к исходной транзакции с помощью *транзакций актуализации* (refresh transaction), которые передаются узлам реплик спустя некоторое время после фиксации транзакции обновления. Транзакция актуализации содержит последовательность обновлений соответствующей транзакции обновления.

Ленивое распространение применяется в приложениях, для которых сильная взаимная согласованность является необязательным и слишком сильным ограничением. Такие приложения могут пожертвовать полной согласованностью реплик в обмен на более высокую производительность. В качестве примеров назовем службу доменных имен (DNS), базы данных, размещенные в далеко отстоящих друг от друга узлах, мобильные базы данных и базы данных персонального цифрового помощника. В этих случаях, как правило, обеспечивается только слабая взаимная согласованность.

Основное преимущество ленивого распространения обновлений – меньшее время ответа, поскольку транзакция обновления может быть зафиксирована сразу после обновления одной копии. Недостатки же в том, что реплики могут быть взаимно не согласованы, некоторые реплики могут устареть, и потому операция локального чтения может прочитать и вернуть неактуальные данные. Кроме того, в некоторых ситуациях, которые мы обсудим ниже, транзакции могут не видеть результатов собственных операций записи, т. е. $R_i(x)$ транзакции обновления T_i может не увидеть результата ранее

выполненной $W_i(x)$. Это явление называется *инверсией транзакции*. Строгая сериализуемость как в одной копии (строгая 1SR) и строгая изоляция моментальных снимков (строгая SI) предотвращают все инверсии транзакций на уровнях изоляции 1SR и SI соответственно, но это достается дорогой ценой. Более слабые гарантии 1SR и глобальная SI обходятся гораздо дешевле, но не предотвращают инверсии транзакций. Были предложены сеансовые транзакционные гарантии на уровнях изоляции 1SR и SI, которые устраняют этот недостаток, предотвращая инверсии транзакций в сеансе клиента, но не между сеансами. Такие сеансовые гарантии обходятся дешевле, чем строгие, но при этом обладают многими желательными свойствами последних.

6.2.3. Централизованные методы

Централизованные методы распространения обновлений требуют, чтобы обновления сначала применялись к главной копии, а затем распространялись на другие (*подчиненные*). Узел, в котором размещена главная копия, называется *главным узлом*, а узлы, содержащие подчиненные копии, – *подчиненными узлами*.

Есть методы, в которых для всех реплицированных данных имеется единственная главная копия. Мы называем их централизованными методами *с одним главным*. В других методах главная копия может зависеть от элемента данных (т. е. для элемента x главная копия x_i находится в узле S_i, а главная копия y_j элемента y_i находится в узле S_j). Это централизованные методы *с ведущей копией* (primary copy).

У централизованных методов два преимущества. Во-первых, применить обновления легко, потому что они производятся только в главном узле, и синхронизация с несколькими узлами реплик не требуется. Во-вторых, есть уверенность, что по крайней мере один узел – тот, в котором размещена главная копия, – содержит актуальные значения элемента данных. Вообще говоря, такие протоколы удобны для хранилищ данных и других приложений, где обработка данных сосредоточена в одном или нескольких главных узлах.

Основной недостаток, как и во всех централизованных алгоритмах, – тот факт, что один центральный узел, где хранятся главные копии всех данных, может оказаться перегружен и станет узким местом. Распределение обязанностей главного узла для каждого элемента данных, как в методах с ведущей копией, – один из способов снизить эти издержки, но тогда возникают проблемы согласованности, особенно в части обеспечения глобальной сериализуемости в ленивых методах репликации, поскольку транзакции актуализации необходимо выполнять на репликах в одном и том же порядке сериализации. Мы обсудим эти вопросы в последующих разделах.

6.2.4. Распределенные методы

В распределенных методах обновление применяется к локальной копии в узле, инициировавшем транзакцию обновления, а затем обновления распространяются на другие узлы реплик. Такие методы называются распре-

деленными, потому что разные транзакции могут обновлять копии одного и того же элемента данных, находящиеся в разных узлах. Они хороши для коллаборативных приложений с распределенными центрами принятия решений и оперативного управления. Они позволяют более равномерно распределить нагрузку и могут обеспечить системе очень высокую доступность, если используются в сочетании с ленивыми методами распространения.

Серьезная трудность возникает в таких системах из-за того, что разные реплики элемента данных могут обновляться в разных узлах (главных узлах) конкурентно. Если распределенный метод сочетается с энергичным распространением, то распределенные алгоритмы управления конкурентностью могут корректно решить проблему конкурентных обновлений. Но если используется ленивое распространение, то транзакции могут выполняться в разном порядке в разных узлах, что порождает глобальную историю, не отвечающую критерию 1SR. К тому же разные реплики могут рассинхронизироваться. Чтобы справиться с этими проблемами, применяется метод урегулирования, включающий отмену и повторное выполнение транзакций таким образом, чтобы порядок выполнения во всех узлах был одинаков. Это не просто, потому что способ урегулирования в общем случае зависит от приложения.

6.3. Протоколы репликации

В предыдущем разделе мы обсудили два признака, по которым классифицируются методы управления обновлениями. Эти признаки независимы, поэтому всего возможно четыре комбинации: энергичная централизованная, энергичная распределенная, ленивая централизованная и ленивая распределенная. В этом разделе мы обсудим перечисленные варианты. Для простоты изложения предполагается, что база данных полностью реплицирована, т. е. все транзакции обновления глобальны. Далее предполагается, что в каждом узле реализован алгоритм управления конкурентностью на основе 2PL.

6.3.1. Энергичные централизованные протоколы

В этом случае главный узел управляет операциями над элементом данных. Эти протоколы сочетаются с методами сильной согласованности, так что обновления логического элемента данных применяются ко всем его репликам в контексте транзакции обновления, которая фиксируется с помощью протокола 2PC (хотя существуют и альтернативы, не опирающиеся на 2PC, которые мы вскоре обсудим). Следовательно, после завершения транзакции обновления во всех репликах значения обновленных элементов данных будут одинаковы (т. е. реплики взаимно согласованы), и результирующая глобальная история удовлетворяет критерию 1SR.

Ранее мы упоминали два параметра, определяющих конкретную реализацию энергичных централизованных протоколов репликации: место выполнения обновлений и степень прозрачности репликации. Первый параметр

(см. раздел 6.2.3) говорит о том, существует ли единственный главный узел для всех элементов данных или разные узлы для каждого элемента либо (что более вероятно) для групп элементов (ведущая копия). Второй параметр говорит о том, должно ли каждое приложение знать о местоположении главной копии (ограниченная прозрачность репликации) или может положиться на свой локальный диспетчер транзакций, который сам найдет главную копию (полная прозрачность репликации).

6.3.1.1. Единственный главный узел с ограниченной прозрачностью репликации

Простейший случай – когда имеется единственная главная копия всей базы данных (т. е. одна для всех элементов данных) с ограниченной прозрачностью репликации, так что все пользовательские приложения знают о главном узле. В этом случае глобальные транзакции обновления (т. е. содержащие по крайней мере одну операцию $W(x)$, где x – реплицированный элемент данных) передаются непосредственно главному узлу, а точнее ДТ в главном узле. В главном узле любая операция чтения $R(x)$ применяется к главной копии (т. е. $R(x)$ преобразуется в $R(x_M)$, где M обозначает главную копию) и выполняется следующим образом: захватить блокировку чтения на x_M, прочитать данные и вернуть результат пользователю. Аналогично любая операция $W(x)$ вызывает обновление главной копии [т. е. выполняется как $W(x_M)$] – получить блокировку записи и выполнить запись. Затем главный ДТ переадресует операцию `Write` подчиненным узлам в синхронном или отложенном режиме (рис. 6.1). В любом случае важно распространять обновления, так чтобы конфликтующие обновления выполнялись в подчиненных узлах в том же порядке, в каком они выполняются в главном узле. Этого можно добиться, применяя временные метки или еще какую-то схему упорядочения.

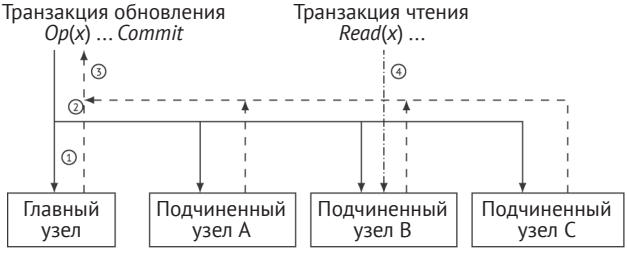


Рис. 6.1 ❖ Действия в протоколе энергичной репликации с единственной главной копией: (1) операция `Write` применяется к главной копии; (2) `Write` распространяется на другие реплики; (3) обновление становится постоянным в момент фиксации; (4) операция `Read` транзакции чтения выполняется над подчиненной копией

Пользовательское приложение может передать транзакцию чтения (в которой имеются только операции `Read`) любому подчиненному узлу. Выполнение транзакций чтения в подчиненных узлах можно производить, следуя

централизованным алгоритмам управления конкурентностью, например C2PL (алгоритмы 5.1–5.3), в которых централизованный диспетчер блокировок находится в узле главной реплики. Реализации на основе C2PL требуют минимальных изменений ДТ в неглавных узлах, в основном для обработки операций `Write`, как описано выше, и их последствий (например, обработки команды `Commit`). Таким образом, когда подчиненный узел получает операцию `Read` (от транзакции чтения), он переадресует ее главному узлу для получения блокировки чтения. Затем операцию `Read` можно выполнить в главном узле и вернуть результат приложению. Или же главный узел может просто послать сообщение «блокировка захвачена» инициировавшему узлу, а тот выполнит `Read` в локальной копии.

Нагрузку на главный узел можно уменьшить, выполнив `Read` в локальной копии без получения блокировки чтения от главного узла. Не важно, используется синхронное или отложенное распространение, локальный алгоритм управления конкурентностью гарантирует, что конфликты чтение-запись должным образом сериализуются, а поскольку операции `Write` могут поступить только от главного узла как часть процедуры распространения обновления, то локальные конфликты запись-запись не могут возникнуть, потому что распространенные транзакции выполняются в каждом узле в порядке, который диктует главный узел. Однако операция `Read` может читать элементы данных в подчиненном узле как до выполнения обновления в нем, так и после. Может случиться, что `Read` из одной транзакции в одном узле прочитает значение одной реплики до обновления, а другая операция `Read` из другой транзакции прочитает другую реплику в другом узле после того же самого обновления, но с точки зрения обеспечения глобальных 1SR-историй это несущественно. Продемонстрируем этот случай на следующем примере.

Пример 6.3. Рассмотрим элемент данных x, для которого главным является узел A, а подчиненными – узлы B и C. Рассмотрим следующие три транзакции:

T_1:	`Write(`x`)`	T_2:	`Read(`x`)`	T_3:	`Read(`x`)`
	`Commit`		`Commit`		`Commit`

Предположим, что T_2 передана узлу B, а T_3 – узлу C. Предположим также, что T_2 читает x в B $[R_2(x_B)]$ до того, как обновление из T_1 применено в B, а T_3 читает x в C $[R_3(x_C)]$ после применения этого обновления в C. Тогда истории, сгенерированные в двух подчиненных узлах, выглядят следующим образом:

$H_B = \{R_2(x), C_2, W_1(x), C_1\}$;
$H_C = \{W_1(x), C_1, R_3(x), C_3\}$.

В узле B порядок сериализации $T_2 \rightarrow T_1$, а в узле C – $T_1 \rightarrow T_3$. Поэтому глобальный порядок сериализации $T_2 \rightarrow T_1 \rightarrow T_3$, и это нормально. Так что история удовлетворяет критерию 1SR. ♦

Следовательно, если придерживаться этого подхода, то транзакции чтения могут читать данные, которые конкурентно обновляются в главном узле, и глобальная история все равно будет отвечать критерию 1SR.

В альтернативном протоколе подчиненный узел S_i, получив операцию $R(x)$, захватывает локальную блокировку чтения, читает из своей локальной копии и возвращает результат пользовательскому приложению; операция может поступить только из транзакции чтения. Получив операцию $W(x)$, поступившую от главного узла, он выполняет ее в своей локальной копии [т. е. $W_i(x_i)$]. Если же эта операция поступила от пользовательского приложения, то он отвергает $W(x)$, поскольку это очевидная ошибка – ведь транзакции обновления должны быть переданы главному узлу.

Эти варианты энергичного централизованного протокола с единственной главной копией легко реализовать. Важный вопрос – как понять, что перед нами: транзакция чтения или транзакция обновления. Его можно решить явным объявлением в команде `Begin_transaction`.

6.3.1.2. Единственный главный узел с полной прозрачностью репликации

Энергичные централизованные протоколы с единственной главной копией требуют от приложения знания главного узла и сильно нагружают главный узел, поскольку он должен обрабатывать (по крайней мере) операции `Read` внутри транзакций обновления, а также играть роль координатора для этих транзакций при выполнении 2PC. В какой-то мере эти вопросы можно решить, если вовлечь в выполнение транзакций обновления ДТ в узле, на котором запущено приложение. Тогда транзакции обновления будут передаваться не главному узлу, а ДТ в том узле, где работает приложение (поскольку им не нужно знать, какой узел главный). Этот ДТ может действовать как координатор для транзакций обновления и чтения. Приложения могут просто передавать свои транзакции локальному ДТ, обеспечив полную прозрачность.

Существуют другие способы реализовать полную прозрачность – координирующий ДТ может действовать только как «маршрутизатор», переправляя каждую операцию напрямую главному узлу. Затем главный узел выполняет операции локально (как описано выше) и возвращает результаты приложению. Этот вариант обеспечивает полную прозрачность, и его просто реализовать, но он не решает проблему перегрузки главного узла. Альтернатива может выглядеть так:

1) координирующий ДТ передает все получаемые операции центральному (главному) узлу. Это не требует внесения каких-либо изменений в алгоритм C2PL-TM (алгоритм 5.1);
2) если это операция $R(x)$, то централизованный диспетчер блокировок (C2PL-LM в алгоритме 5.2) может продолжить работу, захватив блокировку чтения на свою копию x (назовем ее x_M) от имени этой транзакции и проинформировав координирующий ДТ, что блокировка получена. Затем координирующий ДТ может переправить $R(x)$ любому подчиненному узлу, который хранит реплику [т. е. преобразовать ее в $R(x_i)$]. Далее операция чтения выполняется процессором данных (ПД) в этом узле;
3) если это операция $W(x)$, то централизованный диспетчер блокировок (в главном узле) продолжает следующим образом:

a) сначала захватывает блокировку записи на свою копию x_M;
b) затем вызывает свой локальный ПД для применения $W(x_M)$ к своей копии;
c) наконец, информирует координирующий ДТ о том, что блокировка записи получена.

В этом случае координирующий ДТ отправляет $W(x)$ всем подчиненным узлам, на которых хранится копия x, а ПД в этих узлах применяют операцию `Write` к своим локальным копиям.

Фундаментальное отличие этого случая заключается в том, что главный узел не занимается ни операциями `Read`, ни координацией обновлений на репликах. Все это препоручается ДТ узла, в котором работает приложение.

Как легко видеть, этот алгоритм гарантирует, что истории удовлетворяют критерию 1SR, поскольку порядок сериализации определяется единственным главным узлом (как в централизованных алгоритмах управления конкурентностью). Также ясно, что алгоритм следует рассмотренному выше протоколу ROWA – коль скоро гарантируется, что все копии актуальны после завершения транзакции обновления, операцию `Read` можно выполнять над любой копией.

Для демонстрации того, как энергичные алгоритмы сочетают управление репликами с управлением конкурентностью, продемонстрируем алгоритм управления транзакциями для координирующего ДТ (алгоритм 6.1) и алгоритм управления блокировками для главного узла (алгоритм 6.2). Мы показываем только изменения в централизованных алгоритмах 2PL (алгоритмы 5.1 и 5.2 из главы 5).

Алгоритм 6.1. Модификация C2PL-TM для реализации энергичного распространения с единственной главной копией

```
begin
    .
    .
    .
    if запрос блокировки удовлетворен then
        if op.Type = W then
            S ← множество всех подчиненных узлов, где хранится копия
            элемента данных
        else
            S ← какой-нибудь узел, где имеется копия элемента данных
        end if
        DP_S(op)                          {отправить операцию всем узлам в S}
    else
        проинформировать пользователя о завершении транзакции
    end if
    .
    .
    .
end
```

Алгоритм 6.2. Модификация C2PL-LM для реализации энергичного распространения с единственной главной копией

```
begin
    .
    .
    .
    switch op.Type do
        case R or W do                {запрос блокировки; можно ли удовлетворить?}
            найти единицу блокировки lu такую, что op.arg ⊆ lu;
            if lu не заблокирована или режим блокировки lu совместим
              с op.Type then
                захватить блокировку на lu в соответствующем режиме от имени
                транзакции op.tid;
                if op.Type = W then
                    DP_M(op)          {вызвать локальный ПД (M – главный),
                                                          указав операцию}
                отправить сообщение «Блокировка захвачена» ДТ,
                координирующему транзакцию
            else
                поместить op в очередь к lu
            end if
        end case
        .
        .
        .
    end switch
end
```

Заметим, что в приведенных фрагментах алгоритмов диспетчер блокировок посылает назад только сообщение «Блокировка захвачена», а не результат операции обновления. Следовательно, когда координирующий ДТ перешлет обновление подчиненным узлам, они должны будут выполнить его самостоятельно. Иногда это называется *передачей операции*. Можно было бы вместо этого включить в сообщение «Блокировка захвачена» результат вычисления обновления, который затем был бы передан подчиненным узлам, и им осталось бы только применить результат и обновить свои журналы. Это называется *передачей состояния*. Различие может показаться тривиальным, когда операция представлена в виде $W(x)$, но напомним, что эта запись операции `Write` – всего лишь абстракция; в действительности для выполнения каждой операции, возможно, требуется вычислять SQL-выражение, и тогда различие становится весьма существенным.

Описанная выше реализация протокола снимает часть нагрузки с главного узла и не требует, чтобы пользовательские приложения знали, какой узел главный. Однако она сложнее первого рассмотренного варианта. В частности, теперь ДТ в узле, который инициирует транзакции, должен играть роль координатора 2PC, а главный узел становится участником. Поэтому потребуется аккуратно переработать алгоритмы, работающие в этих узлах.

6.3.1.3. Ведущая копия с полной прозрачностью репликации

Теперь ослабим условие единственности главной копии для всех элементов данных – пусть у каждого элемента может быть своя главная копия. В таком случае одна из реплик каждого реплицированного элемента данных назначается *ведущей копией*. Теперь нет одного главного узла, который задал бы глобальный порядок сериализации, поэтому нужно действовать аккуратнее. В полностью реплицированных базах данных любая реплика может быть ведущей копией элемента данных, но если база данных реплицирована частично, то ограниченная прозрачность репликации имеет смысл, только если транзакция обновления обращается лишь к данным, для которых ведущие узлы совпадают. В противном случае прикладная программа не сможет переправлять транзакции обновления одному главному узлу, а должна будет рассматривать каждую операцию в отдельности; кроме того, не понятно, узел какой ведущей копии должен играть роль координатора при выполнении 2PC. Разумной альтернативой является полная прозрачность репликации, когда ДТ в узле приложения действует как координирующий ДТ и переправляет все операции ведущему узлу элемента данных, над которым производится операция. На рис. 6.2 изображена последовательность операций в этом случае, где предыдущее предположение о полной репликации ослаблено. Узел A является главным для элемента данных *x*, а в узлах B и C хранятся реплики (т. е. они подчиненные); аналогично, для элемента *y* главным является узел C, а подчиненными – узлы B и D.

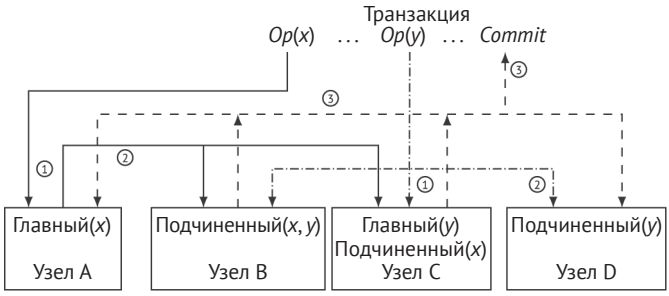


Рис. 6.2 ❖ Действия в протоколе энергичной репликации с ведущей копией:
1) операции (`Read` или `Write`) для каждого элемента данных переправляются главному узлу этого элемента, и `Write` сначала применяется в главном узле;
2) затем `Write` распространяется на другие реплики;
3) обновления становятся постоянными в момент фиксации

Напомним, что в этом варианте обновления по-прежнему применяются ко всем репликам до завершения транзакции, что требует интеграции с методами управления конкурентностью. Самым ранним предложением был алгоритм *двухфазной блокировки с ведущей копией* (primary copy two-phase locking – PC2PL), реализованный в прототипе распределенной версии INGRES. PC2PL – прямое обобщение протокола с единственной главной копией, преследующее цель преодолеть проблемы с производительностью последнего.

Идея в том, чтобы разместить диспетчеры блокировок в нескольких узлах и возложить на каждый ответственность за управление блокировками для заданного множества единиц блокировки, по отношению к которому он является главным узлом. Тогда диспетчеры транзакций будут посылать запросы блокировки и разблокировки диспетчерам, отвечающим за конкретную единицу блокировки. Таким образом, алгоритм трактует одну копию каждого элемента данных как ведущую.

Будучи комбинацией методов управления репликами и управления конкурентностью, подход с ведущей копией нуждается в более сложно устроенном каталоге в каждом узле, но при этом улучшает ранее рассмотренные подходы, снижая нагрузку на главный узел без чрезмерного объема передачи данных между диспетчерами блокировок и транзакций.

6.3.2. Энергичные распределенные протоколы

В случае энергичного распределенного управления репликами обновления могут инициироваться где угодно и сначала применяются к локальной реплике, а затем распространяются на другие. Если обновление инициировано в узле, где не существует реплики элемента данных, то оно переправляется узлу какой-нибудь другой реплики, который координирует выполнение. Все это по-прежнему делается в контексте транзакции обновления, а после ее фиксации обновления становятся постоянными и пользователю отправляется уведомление. На рис. 6.3 изображена последовательность операций для одного логического элемента данных *x* с тремя копиями в узлах A, B, C, D, когда две транзакции обновляют две разные копии (в узлах A и D).

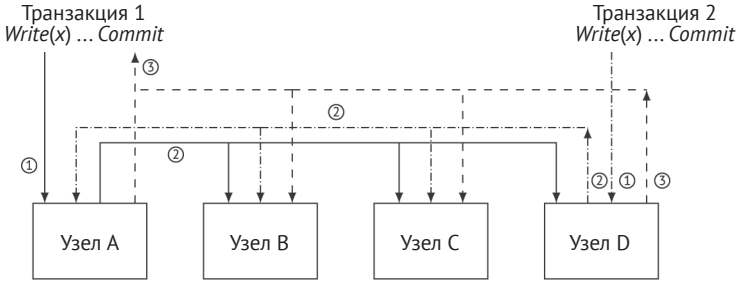


Рис. 6.3 ❖ Действия в энергичном распределенном протоколе репликации: 1) две операции `Write` применяются к двум локальным репликам одного и того же элемента данных; 2) операции `Write` независимо распространяются на другие реплики; 3) обновления становятся постоянными в момент фиксации (показано только для транзакции 1)

Как легко видеть, главная проблема заключается в том, чтобы гарантировать, что конкурентные конфликтующие операции `Write`, инициированные в разных узлах, выполняются в одном и том же порядке во всех узлах, где они выполняются вместе (разумеется, локальное выполнение в каждом узле тоже должно быть сериализовано). Это достигается с помощью методов

управления конкурентностью, реализованных в каждом узле. Следовательно, операции чтения можно выполнять над любой копией, но операции записи выполняются во всех копиях в границах транзакции (например, с помощью алгоритма ROWA) по протоколу управления конкурентностью.

6.3.3. Ленивые централизованные протоколы

Ленивые централизованные протоколы репликации похожи на энергичные тем, что обновления сначала применяются к главной реплике, а затем распространяются на подчиненные. Существенное различие заключается в том, что распространение производится не в контексте транзакции обновления, а после того, как эта транзакция завершится, – в отдельной транзакции актуализации. Следовательно, если подчиненный узел выполняет операцию $R(x)$ над своей локальной копией, то он может прочитать устаревшие (неактуальные) данные, поскольку x, возможно, уже обновлен в главном узле, но до подчиненных обновление еще не дошло.

6.3.3.1. Единственный главный узел с ограниченной прозрачностью репликации

В этом случае транзакции обновления передаются главному узлу и там же выполняются (как в случае энергичного протокола с единственным главным узлом). После того как транзакция будет зафиксирована, подчиненным узлам посылается транзакция актуализации. Последовательность шагов выполнения такова: 1) сначала транзакция обновления применяется к главной реплике; 2) эта транзакция фиксируется в главном узле; 3) подчиненным узлам посылается транзакция актуализации (рис. 6.4).

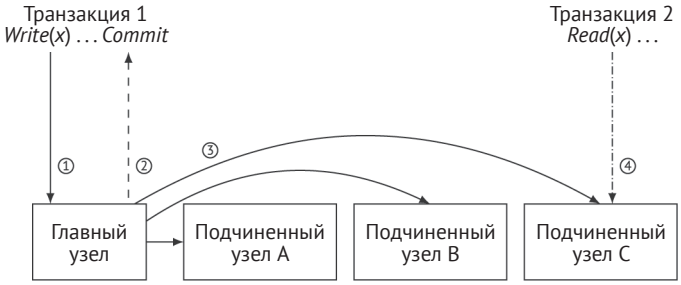


Рис. 6.4 ❖ Действия в ленивом протоколе репликации с одной главной копией:
1) обновление применяется к локальной реплике;
2) после фиксации транзакции обновления становятся постоянными в главном узле;
3) обновление распространяется на другие реплики с помощью транзакций актуализации;
4) транзакция 2 читает из локальной копии

Когда подчиненный узел получает операцию $R(x)$, он читает данные из своей локальной копии и возвращает результат пользователю. Заметим, что, как было отмечено выше, его собственная копия может быть неактуальной, если

главный узел еще обновляется и подчиненный не получил и не выполнил соответствующую транзакцию актуализации. Операция $W(x)$, полученная подчиненным узлом, отвергается (а транзакция отменяется), потому что она должна была быть передана непосредственно главному узлу. Получив транзакцию актуализации от главного узла, подчиненный применяет обновления к своей локальной копии. Получив команду `Commit` или `Abort` (`Abort` может иметь место только для локальных транзакций чтения), подчиненный узел выполняет эти действия локально.

Случай ведущей копии с ограниченной прозрачностью аналогичен, поэтому не будем на нем долго задерживаться. Вместо единственного главного узла операция $W(x)$ передается ведущей копии x, все остальное очевидно.

Как гарантировать применение транзакций актуализации на всех подчиненных узлах в одном и том же порядке? В этой архитектуре существует единственная главная копия для всех элементов данных, поэтому порядок можно определить просто с помощью временных меток. Главный узел должен присоединить временную метку к каждой транзакции актуализации в порядке фиксации исходных транзакций обновления, а подчиненные будут применять транзакции актуализации в порядке следования временных меток.

Аналогичный подход годится и для ведущей копии с ограниченной прозрачностью. В этом случае узел содержит подчиненные копии ряда элементов данных, поэтому получает транзакции актуализации от нескольких главных узлов. Выполнение этих транзакций актуализации должно быть одинаково упорядочено во всех участвующих подчиненных узлах, чтобы состояние базы данных сходилось в конечном счете. Достичь этого можно несколькими способами.

Один из вариантов – назначать временные метки, так чтобы транзакции актуализации, исходящие от разных главных узлов, имели разные метки (добавлять идентификатор узла к монотонно возрастающему счетчику в каждом узле). Тогда транзакции актуализации в каждом узле можно будет выполнять в порядке следования временных меток. Однако транзакции, приходящие не по порядку, вызывают трудности. В традиционных методах на основе временных меток, описанных в главе 5, эти транзакции были бы отменены; но при ленивой репликации это невозможно, потому что транзакция уже была зафиксирована в узле ведущей копии. Единственная возможность – выполнить компенсирующую транзакцию (которая отменяет предыдущую, откатывая ее результаты) или произвести урегулирование обновления – процедуру, которую мы обсудим ниже. Проблему можно решить путем более тщательного изучения результирующих историй, например воспользоваться подходом на основе графа сериализации, при котором строится *граф репликации*, в вершинах которого расположены транзакции (T) и узлы (S), а ребро $\langle T_i, S_j \rangle$ существует тогда и только тогда, когда T_i выполняет операцию `Write` в (реплицированной) физической копии, хранящейся в S_j. Когда передается операция (op_k), в граф репликации вставляются соответствующие вершины (T_k) и ребра, а затем проверяется, нет ли в графе циклов. Если циклов нет, то выполнение можно продолжить. Если же обнаружен цикл, включающий транзакцию, которая была зафиксирована в главном узле, но соответствующие транзакции актуализации еще не зафиксированы во всех релевантных

подчиненных узлах, то текущая транзакция (T_k) отменяется (и перезапускается позже), поскольку ее выполнение породило бы историю, не отвечающую критерию 1SR. В противном случае T_k может подождать завершения остальных транзакций, входящих в цикл (когда они будут зафиксированы в своих главных узлах, а их транзакции актуализации – во всех подчиненных узлах). Когда транзакция завершается подобным манером, соответствующая вершина и все инцидентные ей ребра удаляются из графа репликации. Доказано, что такой протокол порождает истории, отвечающие критерию 1SR. Важный вопрос – поддержание графа репликации. Если он поддерживается одним узлом, то алгоритм становится централизованным. Оставляем построение и поддержание графа репликации в качестве упражнения.

Другой вариант – положиться на механизм групповой передачи данных, предлагаемый базовой инфраструктурой связи (если он поддерживается). Мы обсудим этот вариант в разделе 6.4.

Напомним (см. раздел 6.3.1), что в случае частично реплицированных баз данных энергичный подход с ведущей копией и частичной прозрачностью репликации имеет смысл, если транзакции обновления обращаются только к элементам данных с одним и тем же главным узлом, поскольку эти транзакции целиком выполняются в главном узле. Та же проблема существует в случае ленивой репликации с ведущей копией и частичной прозрачностью. В обоих случаях встает один и тот же вопрос: как спроектировать распределенную базу данных, чтобы можно было выполнять осмысленные транзакции? Эта проблема исследовалась в контексте ленивых протоколов, и был предложен алгоритм выбора ведущего узла, который при заданном наборе транзакций, наборе узлов и наборе элементов данных находит такое назначение ведущих узлов этим элементам (если оно существует), что при выполнении набора транзакций порождается глобальная история, удовлетворяющая критерию 1SR.

6.3.3.2. Единственный главный или ведущий узел с полной прозрачностью репликации

Теперь обратимся к алгоритмам, которые обеспечивают полную прозрачность, разрешая передавать транзакции (чтения или обновления) любому узлу и переправлять их операции либо единственному главному узлу, либо подходящему ведущему узлу. Это непросто и требует решения двух проблем: во-первых, при недостаточном внимании не гарантируется порождение глобальной истории, отвечающей критерию 1SR, а во-вторых, транзакция может не увидеть собственных обновлений. Продемонстрируем обе проблемы на примерах.

Пример 6.4. Рассмотрим случай единственного главного узла, когда узел M содержит главные копии *x* и *y*, а узел B – их подчиненные копии. Пусть есть две транзакции: T_1 инициирована в узле B, а T_2 – в узле M:

```
T1: Read(x)        T2: Write(x)
    Write(y)           Write(y)
    Commit             Commit
```

Выполнить их при соблюдении полной прозрачности можно было бы, например, следующим образом. T_2 следует выполнить в узле M, поскольку тот содержит главные копии *x* и *y*. Спустя какое-то время после ее фиксации транзакции актуализации для входящих в нее операций `Write` посылаются узлу B для обновления подчиненных копий. С другой стороны, T_1 могла бы прочитать локальную копию *x* в узле B [$R_1(x_B)$], но входящая в нее операция $W_1(x)$ должна быть переправлена главной копии *x*, т. е. узлу M. Спустя некоторое время после выполнения и фиксации $W_1(x)$ в главном узле узлу B будет послана транзакция актуализации для обновления подчиненной копии. Ниже показана возможная последовательность шагов выполнения (рис. 6.5):

1) $R_1(x)$ инициирована в узле B, где и выполнена [$R_1(x_B)$];
2) $W_2(x)$ инициирована в узле M и выполнена [$W_2(x_M)$];
3) $W_2(y)$ инициирована в узле M и выполнена [$W_2(y_B)$];
4) T_2 инициирует свою операцию `Commit` в узле M, и там же она и выполняется;
5) $W_1(x)$ инициирована в узле B; поскольку главная копия *x* находится в узле M, операция `Write` переправляется на M;
6) $W_1(x)$ выполняется в узле M [$W_1(x_M)$], подтверждение передается назад узлу B;
7) T_1 инициирует свою операцию `Commit` в узле B, она переправляется узлу M, там выполняется, узел B получает уведомление о фиксации и тоже фиксирует T_1;
8) узел M посылает транзакцию актуализации для T_2 узлу B, где она выполняется и фиксируется;
9) наконец, узел M посылает транзакцию актуализации для T_1 узлу B (она содержит входящую в T_1 операцию `Write`, которая была выполнена в главном узле), она выполняется и фиксируется в B.

В двух узлах генерируются следующие истории (верхний индекс *r* при операциях означает, что они являются частью транзакции актуализации):

$H_B = \{W_2(x_M), W_2(y_M), C_2, W_1(y_M), C_1\}$;
$H_B = \{R_1(x_B), C_1, W_2^r(x_B), W_2^r(y_B), C_2^r, W_1^r(x_B), C_1^r\}$.

Результирующая глобальная история для *логических* элементов данных *x* и *y* не удовлетворяет критерию 1SR. ♦

Пример 6.5. Снова рассмотрим сценарий с единственной главной копией, в котором узел M содержит главную копию *x*, а узел D – ее подчиненную копию. Пусть имеется следующая простая транзакция:

T_3: `Write`(*x*)
`Read`(*x*)
`Commit`

Следуя той же модели выполнения, что в примере 6.4, мы можем записать такую последовательность шагов:

1) $W3(x)$ инициирована в узле D, который переправляет ее узлу M для выполнения;
2) операция `Write` выполняется в M [$W_3(x_M)$], подтверждение посылается узлу D;

3) $R_3(x)$ инициирована в узле D и выполняется над локальной копией $[R_3(x_D)]$;
4) T_3 инициирует фиксацию в D, она переправляется узлу M, там выполняется, а уведомление посылается узлу D, который также фиксирует транзакцию;
5) узел M посылает узлу D транзакцию актуализации для операции $W_3(x)$;
6) узел D выполняет и фиксирует транзакцию актуализации.

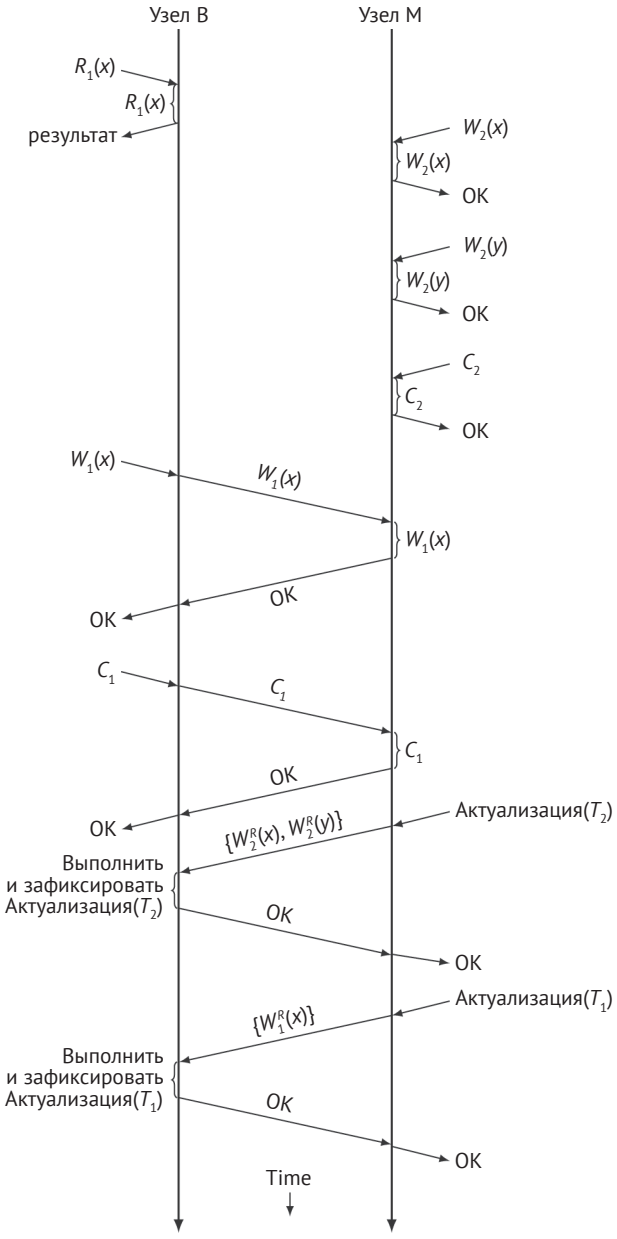


Рис. 6.5 ❖ Временная последовательность выполнения транзакций

Заметим, что транзакция актуализации посылается узлу D спустя некоторое время после того, как T_3 была зафиксирована в M, поэтому на шаге 3 она читает старое значение x в узле D и не видит результата собственной операции `Write`, предшествующей `Read`. ♦

Из-за этих проблем существует не так много предложений по реализации полной прозрачности в ленивых алгоритмах репликации. Исключение представляет алгоритм, в котором рассматривается случай единственной главной копии и предлагается метод проверки корректности главным узлом в точке фиксации, похожий на оптимистическое управление конкурентностью. Основная идея состоит в следующем. Рассмотрим транзакцию T, которая записывает элемент данных x. В момент фиксации T главный узел генерирует временную метку и использует ее для пометки главной копии (x_M), которая запоминает временную метку последней обновившей ее транзакции (*last_modified*(x_M)). Эта метка добавляется и в транзакции актуализации. Подчиненные узлы, получив транзакцию актуализации, записывают в свои копии то же значение, т. е. *last_modified*(x_i) ← *last_modified*(x_M). Генерирование временных меток для T в главном узле подчиняется следующему правилу:

> Временная метка для транзакции T должна быть больше всех ранее сгенерированных временных меток и меньше, чем временные метки *last_modified* элементов данных, к которым она обращается. Если сгенерировать такую временную метку невозможно, то транзакция T отменяется[1].

Эта проверка гарантирует, что операции чтения читают правильные значения. Так, в примере 6.4 главный узел M не смог бы корректно назначить временную метку транзакции T_1 в момент фиксации, т. к. *last_modified*(x_M) отражала бы обновление, произведенное T_2. Поэтому T_1 была бы отменена.

Этот алгоритм решает первую из отмеченных выше проблем, но автоматически не гарантирует, что транзакция будет видеть результаты своих же операций записи (проблема, которую мы выше назвали инверсией транзакции). Для решения этой проблемы предлагалось вести список всех обновлений, произведенных транзакцией, и сверяться с этим списком при выполнении `Read`. Но, поскольку только главный узел знает про обновления, вести список и выполнять все операции `Read` и `Write` пришлось бы на нем.

6.3.4. Ленивые распределенные протоколы

Ленивые распределенные протоколы репликации самые сложные, потому что обновляться может любая реплика, после чего изменения нужно лениво распространить на другие реплики (рис. 6.6).

В узле, где транзакция инициирована, протокол работает просто: операции `Read` и `Write` применяются к локальной копии, и транзакция фиксируется

---

[1] В оригинальном предложении учитывается широкий спектр ограничений на актуальность, которые мы обсуждали выше, поэтому правило сформулировано в более общем виде. Но поскольку нас интересует в основном поведение, отвечающее критерию 1SR, то подойдет и эта (более строгая) формулировка.

локально. Спустя некоторое время после фиксации обновления распространяются на другие узлы с помощью транзакций актуализации.

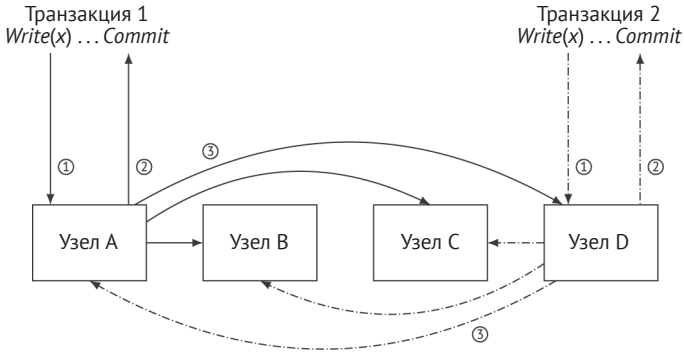


Рис. 6.6 ❖ Действия в ленивом распределенном протоколе репликации:
1) два обновления применены к двум локальным репликам;
2) после фиксации транзакции обновления становятся постоянными;
3) обновления независимо распространяются на другие реплики

Затруднения возникают при обработке обновлений в других узлах. Получив транзакцию обновления, узел должен локально запланировать ее, для чего применяется локальный механизм управления конкурентностью. Для достижения надлежащей сериализации транзакций актуализации можно воспользоваться методами, рассмотренными в предыдущих разделах. Однако несколько транзакций могут конкурентно обновлять разные копии одних и тех же данных в разных узлах, и эти обновления могут конфликтовать друг с другом. Изменения необходимо урегулировать, и это осложняет упорядочение транзакций актуализации. По результатам урегулирования определяется порядок выполнения транзакций актуализации, после чего обновления применяются в каждом узле.

Здесь критическим является вопрос об урегулировании. Можно спроектировать алгоритм урегулирования общего назначения на основе эвристик. Например, применять обновления в порядке временных меток (применяется та, у которой метка позже) или отдавать предпочтение обновлениям, инициированным в определенных узлах (быть может, наиболее важных). Однако все это ситуативные методы, а в действительности урегулирование зависит от семантики приложения. Кроме того, какой метод урегулирования ни выбрать, все равно какие-то обновления теряются. Заметим, что упорядочение на основе временных меток работает, только если локальные часы, используемые для генерирования меток, синхронизированы. Выше мы говорили, что в крупномасштабных распределенных системах добиться этого нелегко. Если просто создавать временную метку, конкатенируя номер узла и показания локальных часов, то предпочтение, отдаваемое определенным транзакциям, может не иметь никаких оснований в логике приложения. Тот факт, что временные метки хорошо работают в алгоритмах управления конкурентностью, но не в этом случае, объясняется просто: для управления конкурентностью

нам нужен *какой-то* порядок, здесь же нас интересует *конкретный* порядок, согласованный с семантикой приложения.

6.4. Групповая коммуникация

В предыдущем разделе мы говорили о том, как велики могут быть издержки протоколов репликации, особенно в плане накладных расходов на передачу сообщений. Опишем очень упрощенную модель стоимости для алгоритмов репликации. Если имеется n реплик и каждая транзакция состоит из m операций обновления, то каждая транзакция генерирует $n * m$ сообщений (если можно организовать многоадресную передачу, то достаточно m сообщений). Если система хочет поддержать пропускную способность k транзакций в секунду, то получится $k * n * m$ сообщений в секунду (или $k * m$ в случае многоадресной передачи). Можно несколько усложнить эту функцию стоимости, включив в рассмотрение время выполнения каждой операции (быть может, зависящее от нагрузки на систему). Проблема многих рассмотренных выше протоколов репликации (в особенности распределенных) – высокие накладные расходы на передачу сообщений.

Для эффективной реализации протоколов очень важно снизить эти накладные расходы. Предлагались решения, в которых используются протоколы групповой коммуникации в сочетании с нетрадиционными методами обработки локальных транзакций. Эти решения вносят две модификации. Во-первых, в них не используется протокол 2PC в момент фиксации, а для достижения согласия применяются протоколы групповой коммуникации. Во-вторых, используется отложенное, а не синхронное распространение обновлений.

Сначала рассмотрим идею групповой коммуникации. Система групповой коммуникации позволяет одному узлу рассылать сообщение всем узлам группы с гарантией доставки, т. е. рано или поздно сообщение будет доставлено всем адресатам. Кроме того, система может предоставлять примитивы многоадресной коммуникации с различным порядком доставки, из которых нам интересен только один: полное упорядочение. В случае вполне упорядоченной групповой коммуникации все сообщения, отправляемые узлом, доставляются в одинаковом порядке всем адресатам. Это важно для понимания изложенного ниже материала.

Мы продемонстрируем использование групповой коммуникации на примере двух протоколов. Первый – альтернатива энергичному распределенному протоколу, второй – ленивый централизованный протокол.

В энергичном распределенном протоколе, основанном на групповой коммуникации, используется локальная стратегия обработки, в которой операции `Write` выполняются над локальными теневыми копиями, находящимися там, где инициирована транзакция, и применяется вполне упорядоченная групповая коммуникация для многоадресной рассылки множества операций записи, входящих в транзакцию, всем остальным репликам. Полная упорядоченность коммуникации гарантирует, что все узлы получат операции записи в одном и том же порядке, а значит, порядок сериализации во всех узлах

будет одинаков. Для простоты изложения мы далее будем предполагать, что база данных полностью реплицирована и что каждый узел реализует алгоритм управления конкурентностью 2PL.

Согласно протоколу, выполнение транзакции T_i состоит из четырех шагов (действия, относящиеся к локальному управлению конкурентностью, не показаны).

I. **Фаза локальной обработки.** Операция $R_i(x)$ выполняется в том узле, на котором инициирована (это главный узел для данной транзакции). Операция $W_i(x)$ также выполняется в главном узле, но над теневой копией (см. обсуждение механизма теневых страниц в предыдущей главе[1]).

II. **Фаза коммуникации.** Если T_i состоит только из операций `Read`, то она может быть зафиксирована в главном узле. Если же она включает операции `Write` (т. е. является транзакцией обновления), то ДТ в главном узле T_i (том, где была инициирована T_i) собирает операции записи в одно *сообщение записи* WM_i[2] и рассылает всем узлам-репликам (в т. ч. себе самому) с применением вполне упорядоченной групповой коммуникации.

III. **Фаза блокировки.** Когда WM_i доставляется в узел S_j, все блокировки запрашиваются разом, в рамках одного атомарного шага. Это можно сделать, поставив защелку (облегченная форма блокировки) на таблицу блокировок, которая удерживается до тех пор, пока не захвачены все блокировки или запросы не поставлены в очередь. Выполняются следующие действия:

1) для каждой операции $W(x)$ в WM_i (будем обозначать x_j копию x, хранящуюся в узле S_j):
   a) если не существует других транзакций, заблокировавших x_j, то захватить блокировку записи на x_j;
   b) в противном случае проверить наличие конфликтов:
      - если существует локальная транзакция T_k, которая уже заблокировала x_j, но она находится в фазе локального чтения или коммуникации, то T_k отменяется. Кроме того, если T_k находится в фазе коммуникации, то всем узлам рассылается сообщение с окончательным решением «отменить». На этой стадии обнаруживаются конфликты чтение-запись и локальные транзакции чтения просто отменяются. Заметим, что лишь локальные операции чтения получают блокировки в фазе локальной обработки, потому что локальные операции записи применяются только к теневым копиям. Поэтому проверять наличие конфликтов запись-запись на этой стадии нет необходимости;
      - в противном случае запрос блокировки $W_i(x_j)$ ставится в очередь к x_j;

---

[1] Ни в предыдущей, ни в какой другой главе теневые страницы не обсуждаются. – *Прим. перев.*

[2] Передаются обновленные элементы данных (т. е. имеет место передача состояния).

2) если T_i – локальная транзакция (напомним, что сообщение посылается также узлу, инициировавшему транзакцию, и в этом случае $j = i$), то узел может зафиксировать ее, поэтому он рассылает группе сообщение «фиксировать». Отметим, что это сообщение посылается сразу после запроса блокировок, а не после записи, т. е. последовательность не такая, как в протоколе 2PC.

IV. **Фаза записи.** Когда узлу удается получить блокировку записи, он применяет соответствующее обновление (в главном узле, и это означает, что теневая копия стала действительной версией). Узел, инициировавший T_i, может зафиксировать ее и освободить все блокировки. Другие узлы ждут сообщения с окончательным решением и поступают соответственно.

В этом протоколе важно сделать так, чтобы фазы блокировки конкурентных транзакций выполнялись в одинаковом порядке в каждом узле; именно эту цель и преследует вполне упорядоченная многоадресная коммуникация. Отметим также, что к сообщениям, содержащим решение (шаг III.2), не предъявляется требований о порядке следования, они могут быть доставлены в любом порядке, даже раньше, чем соответствующее сообщение *WM*. В таком случае узлы, получившие сообщение с решением раньше *WM*, просто регистрируют это решение, но не предпринимают никаких действий. Когда сообщение *WM* придет, они смогут выполнить фазы блокировки и записи и завершить транзакцию, как предписывает ранее доставленное сообщение с решением.

Производительность этого протокола значительно выше, чем у наивного протокола, рассмотренного в разделе 6.3.2. Для каждой транзакции главный узел посылает два сообщения: первое – *WM*, второе – содержащее решение. Поэтому, чтобы обеспечить пропускную способность k транзакций в секунду, общее число сообщений должно быть $2k$, а не $k*m$, как в наивном протоколе (предполагается, что в обоих случаях применяется многоадресная передача). Кроме того, производительность системы увеличивается за счет отложенного энергичного распространения, поскольку синхронизация узлов-реплик для всех операций `Write` производится один раз в самом конце, а не на всем протяжении выполнения транзакции.

Второй пример использования групповой коммуникации мы обсудим в контексте ленивых централизованных алгоритмов. Напомним, что в этом случае важно гарантировать, что транзакции актуализации упорядочены одинаково во всех участвующих узлах, иначе состояния базы данных не сойдутся. Если система поддерживает вполне упорядоченную многоадресную передачу, то транзакции актуализации, отправленные разными главными узлами, будут доставлены всем узлам в одном и том же порядке. Однако полный порядок при передаче сообщений сопровождается высокими накладными расходами, что может ограничить масштабируемость. Требование о поддержании полного порядка системой коммуникации можно ослабить и разрешить протоколу репликации самому отвечать за упорядочение выполнения транзакций актуализации. Мы продемонстрируем этот вариант на примере протокола, в котором используется упорядоченная с помощью FIFO-очереди многоадресная передача данных с ограниченной задержкой

(обозначим ее *Max*), а также предполагается, что часы синхронизированы неточно, но могут расходиться не более чем на ε. Дополнительно предполагается, что в каждом узле имеются средства управления транзакциями. В результате работы протокола репликации в каждом подчиненном узле поддерживается «очередь готовых», где хранится упорядоченный список транзакций актуализации, которые подаются диспетчеру транзакций для локального выполнения. Таким образом, протокол гарантирует, что очереди в каждом подчиненном узле, где выполняются транзакции актуализации, упорядочены одинаково.

В каждом подчиненном узле поддерживается «очередь ожидающих» для каждого главного узла, подчиненным которого является данный узел (т. е. если в подчиненном узле хранятся реплики *x* и *y*, главными узлами которых являются узлы S_1 и S_2, то будет две очереди ожидающих, q_1 и q_2, соответствующие главным узлам *S*1 и *S*2). Когда в главном узле $site_k$ создается транзакция актуализации RT_i^k, ей назначается временная метка $ts(RT_i)$, соответствующая физическому времени в момент фиксации соответствующей транзакции обновления T_i. Когда RT_i приходит в подчиненный узел, она помещается в очередь q_k. При поступлении каждого сообщения просматриваются элементы в начале всех очередей ожидающих, и тот, у которого временная метка наименьшая, выбирается в качестве новой *RT* (*new_RT*), подлежащей обработке. Если значение *new_RT* изменилось с момента последней итерации (т. е. поступила новая *RT* с временной меткой, меньшей, чем та, что была выбрана на предыдущей итерации), то транзакция с меньшей временной меткой становится *new_RT* и рассматривается планировщиком.

Когда транзакция актуализации выбрана в качестве *new_RT*, она не сразу помещается в очередь готовых для диспетчера транзакций; при планировании транзакции актуализации принимается во внимание максимальная задержка и возможная рассинхронизация локальных часов. Это делается для того, чтобы у любой транзакции актуализации, которая может быть задержана, гарантированно появился шанс попасть в подчиненный узел. Время, в течение которого RT_i будет поставлена в очередь готовых в подчиненном узле, равно $delivery\_time = ts(new\_RT) + Max + \varepsilon$. Поскольку система коммуникации гарантирует верхнюю границу *Max* при доставке сообщений и поскольку максимальная рассинхронизация локальных часов (определяющих временные метки) составляет ε, доставка транзакции актуализации на любой подчиненный узел не может быть задержана больше, чем на *delivery_time*. Таким образом, протокол гарантирует, что транзакция актуализации планируется для выполнения в подчиненном узле, когда удовлетворяются следующие условия: (1) все операции записи соответствующей транзакции обновления выполнены в главном узле (2) в порядке, определенном временной меткой транзакции актуализации, и (3) в самый ранний физический момент времени, эквивалентный ее *delivery_time*. Поэтому обновление ведомых копий в подчиненных узлах производится в том же хронологическом порядке, в каком были обновлены их ведущие копии, и этот порядок одинаков для всех участвующих подчиненных узлов в предположении, что базовая инфраструктура коммуникации способна гарантировать *Max* и ε. Это пример ленивого алгоритма, который гарантирует, что глобальная история отвечает

критерию 1SR, но при этом согласованность слабая, т. е. значения реплик могут изменяться не одновременно, а с интервалом, не превышающим заранее заданное время.

6.5. Репликация и отказы

До сих пор мы рассматривали протоколы репликации в отсутствие отказов. А что произойдет со взаимной согласованностью, если системы могут выходить из строя? Отказы обрабатываются по-разному при энергичной и ленивой репликациях.

6.5.1. Отказы и ленивая репликация

Сначала рассмотрим, как обрабатываются отказы при ленивой репликации. Это сравнительно простой случай, потому что в этих протоколах разрешено расхождение между главными копиями и репликами. Следовательно, когда в результате разрыва связи один или несколько узлов становятся недоступны, оставшиеся доступными узлы могут продолжать работу. Даже в случае разделения сети можно разрешить независимую работу в нескольких частях, а после восстановления позаботиться о сходимости состояний базы данных, применив методы разрешения конфликтов, описанные в разделе 6.3.4. До слияния базы данных в разных частях расходились, но в процессе слияния все конфликты будут урегулированы.

6.5.2. Отказы и энергичная репликация

Теперь обратимся к энергичной репликации – этот случай гораздо сложнее. Выше мы говорили, что все энергичные методы реализуют ту или иную разновидность протокола ROWA, гарантирующего, что после фиксации транзакции значения всех реплик будут одинаковы. Семейство протоколов ROWA элегантно и привлекательно. Но, как мы видели при обсуждении протоколов фиксации, у него есть один большой недостаток. Если недоступна хотя бы одна реплика, то транзакцию обновления невозможно завершить. Поэтому ROWA не позволяет достичь одной из фундаментальных целей репликации – обеспечить высокую доступность.

Альтернативой ROWA, разработанной в попытке решить проблему низкой доступности, является протокол *чтение одной / запись всех доступных* (Read-One/Write-All Available – ROWA-A). Идея в том, чтобы выполнить команды записи для всех доступных копий и завершить транзакцию. Копии, которые были недоступны в этот момент, должны будут «подтянуться», когда станут доступны.

Существуют различные версии этого протокола, мы обсудим две из них. Первая называется *протоколом доступных копий* (available copies protocol). Координатор транзакции обновления T_i (т. е. главный узел, в котором выполняется транзакция) посылает каждую операцию $W_i(x)$ всем подчиненным

узлам, где хранятся реплики x, и ждет подтверждения выполнения (или отказа). Если тайм-аут истечет до получения подтверждения от всех узлов, то неответившие узлы считаются недоступными, но работа продолжается – обновление производится в доступных узлах. Заметим, однако, что эти узлы могут даже не знать о существовании T_i и произведенном ей обновлении x, если стали недоступны до начала T_i.

Необходимо разрешить две проблемы. Во-первых, может случиться, что узлы, которые координатор счел недоступными, на самом деле прекрасно работают и уже обновили x, но отправленное ими подтверждение не успело дойти до координатора. Во-вторых, некоторые узлы могли быть недоступны, когда T_i началась, но затем восстановились и стали выполнять транзакции. Поэтому координатор запускает процедуру проверки, перед тем как зафиксировать T_i.

1. Координатор проверяет, все ли узлы, которые он считал недоступными, по-прежнему недоступны. Для этого он посылает сообщение-вопрос каждому из этих узлов. Доступные узлы отвечают. Если координатор получил ответ от одного из таких узлов, то он отменяет T_i, т. к. не знает, в каком состоянии находится ранее недоступный узел: быть может, он все это время был доступен и выполнил операцию $W_i(x)$, но просто задержалось подтверждение (в таком случае все хорошо), а быть может, он действительно был недоступен в момент начала T_i, но впоследствии стал доступен. Возможно даже, что он выполнил $W_j(x)$ от имени другой транзакции T_j. В последнем случае продолжение T_i сделало бы историю выполнения несериализуемой.
2. Если координатор T не получил ответа ни от одного из узлов, которые он считал недоступными, то он проверяет, все ли узлы, которые были доступны в процессе выполнения $W_i(x)$, по-прежнему доступны. Если да, то T можно зафиксировать. Естественно, этот второй шаг можно включить в протокол фиксации.

Второй вариант ROWA-A, который мы обсудим, – распределенный протокол ROWA-A. В этом случае каждый узел S хранит множество узлов V_S, которые он считает доступными; это «представление» S о конфигурации системы. В частности, когда инициируется транзакция T_i, представление ее координатора отражает все узлы, доступность которых не вызывает у координатора сомнения (для простоты обозначим это множество $V_C(T_i)$). Операция $R_i(x)$ выполняется для любой реплики, принадлежащей $V_C(T_i)$, а $W_i(x)$ обновляет все эти реплики. Координатор проверяет свое представление в конце T_i, и если оно изменилось с момента начала T_i, то T_i отменяется. Для модификации V во всех узлах выполняется специальная атомарная транзакция, гарантирующая, что не генерируется никаких конкурентных представлений. Для этого можно назначать каждому V временную метку в момент генерирования и убедиться, что узел принимает новое представление, только если номер его версии больше, чем номер версии текущего представления, хранимого этим узлом.

Класс протоколов ROWA-A более устойчив к отказам, в т. ч. к разделению сети, чем простой протокол ROWA.

Еще один класс энергичных протоколов репликации основан на голосовании. Фундаментальные характеристики голосования были представлены

в предыдущей главе при обсуждении разделения сети в нереплицированных базах данных. Общие идеи остаются в силе и для случая репликации. Суть в том, что каждая операция чтения и записи должна получить достаточное для фиксации количество голосов. Такие протоколы могут быть пессимистическими и оптимистическими. Далее мы будем рассматривать только пессимистические протоколы. В оптимистической версии используются компенсирующие транзакции для восстановления, в случае когда решение о фиксации не подтверждается при завершении. Эта версия подходит, если применение компенсирующих транзакций допустимо (см. главу 5).

Самый ранний из известных алгоритмов голосования (алгоритм Томаса) работает с полностью реплицированными базами данных и назначает всем узлам одинаковое число голосов. Прежде чем выполнить любую операцию транзакции, он должен получить голоса «за» от большинства узлов. Эта идея была пересмотрена в алгоритме Гиффорда, который работает также с частично реплицированными базами данных и назначает голос каждой копии реплицированного элемента данных. Затем каждая операция должна получить *кворум чтения* (V_r) или *кворум записи* (V_w), чтобы соответственно прочитать или записать элемент. Если некоторому элементу данных всего назначено V голосов, то кворумы должны подчиняться следующим правилам:

1) $V_r + V_w > V$;
2) $V_w > V/2$.

Напомним (см. предыдущую главу), что первое правило гарантирует, что к элементу данных одновременно не обращаются транзакции чтения и записи (т. е. не возникает конфликта чтение-запись). С другой стороны, второе правило гарантирует, что две операции записи из двух разных транзакций не могут применяться одновременно к одному и тому же элементу данных (т. е. не возникает конфликта запись-запись). Таким образом, оба правила вместе гарантируют сериализуемость и эквивалентность всех копий.

В случае разделения сети протоколы на основе кворума хорошо работают, потому что, по сути дела, определяют, какие транзакции смогут завершиться, основываясь на голосах, которые они могут получить. Правило выделения голосов и пороговые условия гарантируют, что две транзакции, инициированные в разных частях сети и обращающиеся к одним и тем же данным, не могут завершиться одновременно.

Трудность, с которой сталкивается эта версия протокола, заключается в том, что транзакциям нужно набрать кворум, даже если они хотят только читать данные. Это заметно и без необходимости замедляет чтение из базы данных. Ниже описывается другой протокол голосования на основе кворума, который преодолевает этот серьезный недостаток.

В протоколе делаются некоторые предположения о находящемся ниже уровне коммуникации и возникновении отказов. Предполагается, что отказы «чистые». Это означает две вещи:

1) отказы, изменяющие топологию сети, мгновенно обнаруживаются всеми узлами;
2) у каждого узла имеется свое представление о сети, содержащее все узлы, с которыми он может обмениваться сообщениями.

Если сеть связи удовлетворяет этим двум условиям, то протокол управления репликами сводится к простой реализации принципа ROWA-A. Прежде чем читать или записывать элемент данных, протокол проверяет, находится ли большинство узлов в той же части сети, что узел, на котором он работает. Если да, то протокол реализует правила ROWA внутри этой части: читает любую копию элемента данных и записывает все копии, находящиеся в этой части.

Заметим, что операция чтения или записи выполняется только в одной части сети. Следовательно, это пессимистический протокол, который гарантирует сериализуемость как в одной копии, *но только внутри этой части*. После восстановления связности сети база данных восстанавливается посредством распространения результатов обновления на другие части.

Фундаментальный вопрос по поводу этого протокола – реалистичны ли предположения об отказах. К сожалению, это не всегда так, поскольку большинство сетевых отказов не «чистые». От возникновения отказа до его обнаружения узлом проходит некоторое время. Из-за этой задержки узел может думать, что находится в одной части сети, тогда как на самом деле случившиеся позже отказы помещают его в другую часть. Кроме того, для разных узлов задержка может быть различна. Поэтому два узла, когда-то находившиеся в одной части, а затем оказавшиеся в разных, могут некоторое время продолжать работу в предположении, что они все еще находятся в одной части. Нарушения этих двух предположений об отказах крайне негативно влияют на протокол управления репликами и его способность обеспечить сериализуемость как в одной копии.

Предлагаемое решение – надстроить над физическим уровнем связи еще один уровень абстракции, который скрывает «нечистоту» отказов физического уровня и предоставляет протоколу управления репликами коммуникационную службу, в которой все отказы «чистые». Этот новый уровень абстракции предоставляет *виртуальные разделы*, внутри которых работает протокол управления репликами. Виртуальный раздел – это группа узлов с общим представлением о том, кто находится в этом разделе. Узлы присоединяются к виртуальным разделам и отсоединяются от них под контролем этого нового коммуникационного уровня, гарантирующего соблюдение предположений о чистоте отказов.

Преимущество этого протокола – в его простоте. Нет никаких накладных расходов на собирание кворума для операций чтения. Поэтому чтение может выполняться так же быстро, как в неразделенной сети. К тому же он достаточно общий, так что протоколу управления репликами не нужно различать отказы узлов и разделение сети.

Коль скоро существуют альтернативные методы достижения отказоустойчивости в случае реплицированных баз данных, возникает естественный вопрос: каковы их сравнительные достоинства? Существует немало исследований, в которых эти методы анализируются при различных допущениях. По их результатам можно сделать вывод, что реализации ROWA-A позволяют достичь лучшей масштабируемости и доступности, чем методы на основе кворума.

6.6. Заключение

В этой главе мы обсудили различные подходы к репликации данных и познакомились с протоколами, применяемыми при разных условиях. У каждого из рассмотренных протоколов есть свои плюсы и минусы. Энергичные централизованные протоколы легко реализуются, не требуют координации обновлений в разных узлах и гарантированно приводят к сериализуемым историям как в одной копии. Однако они сильно нагружают главный узел, так что он может стать узким местом. Поэтому их сложнее масштабировать, особенно в архитектуре с единственным главным узлом – варианты с ведущей копией масштабируются лучше, потому что обязанности главного узла в какой-то мере распределены. Для этих протоколов характерно большое время ответа (самое большое среди всех четырех вариантов), поскольку, чтобы обратиться к любому элементу данных, нужно дождаться фиксации всех транзакций, которые в данный момент его обновляют (при этом используется дорогостоящий протокол 2PC). Кроме того, локальные копии используются не в полную силу – только для чтения. Таким образом, если в рабочей нагрузке превалируют операции обновления, то энергичные централизованные протоколы, скорее всего, будут демонстрировать низкую производительность.

Энергичные распределенные протоколы также гарантируют сериализуемость как в одной копии и предлагают элегантное симметричное решение, в котором все узлы функционируют одинаково. Но если система коммуникации не поддерживает эффективную многоадресную передачу, то количество сообщений оказывается очень велико, что увеличивает нагрузку на сеть и приводит к большому времени ответа. Также это ограничивает масштабируемость. Кроме того, наивная реализация этих протоколов чревата большим количеством взаимоблокировок, поскольку операции обновления выполняются в нескольких узлах конкурентно.

У ленивых централизованных протоколов очень короткое время ответа, поскольку транзакции выполняются и фиксируются в главном узле и не должны ждать завершения в подчиненных узлах. Также не нужно координировать выполнение транзакций обновления в разных узлах, что сокращает количество сообщений. С другой стороны, взаимная согласованность (т. е. актуальность всех копий) не гарантируется, потому что локальные копии могут устаревать. Это означает, что нет уверенности в актуальности данных, прочитанных из локальной копии.

Наконец, у ленивых протоколов с несколькими главными узлами время ответа самое короткое, а доступность наивысшая, поскольку все транзакции выполняются локально, без распределенной координации. Лишь после фиксации остальные реплики обновляются с помощью транзакций актуализации. Но это также и недостаток таких протоколов – разные реплики могут быть обновлены разными транзакциями, что требует разработки сложных протоколов урегулирования и может приводить к потере обновлений.

Репликации посвящено много исследований как со стороны специалистов по распределенным вычислениям, так и со стороны специалистов по базам

данных. Хотя постановка задачи в обеих дисциплинах похожа, есть и существенные различия. Назовем два самых важных. Предметом репликации данных являются только данные, тогда как репликация вычислений не менее важна и в распределенных вычислениях. В частности, много внимания было уделено проблемам репликации данных в мобильных средах, где возможна работа без соединения. Во-вторых, согласованность на уровне транзакций и базы данных играет огромную роль в репликации данных, тогда как в распределенных вычислениях вопросы согласованности занимают не первое место в списке приоритетов. Поэтому были определены значительно более слабые критерии согласованности.

Репликация изучалась также в контексте параллельных систем баз данных, в особенности в параллельных кластерах. Мы будем обсуждать их в главе 8. Вопросы репликации в системах управления мультибазами данных мы отложим до главы 7.

6.7. Библиографические замечания

Протоколы репликации и управления репликами были предметом исследований с самого начала изучения распределенных баз данных. Их обзор приведен в работе [Helal et al. 1997]. Обзор протоколов управления репликами в условиях разделения сети имеется в работе [Davidson et al. 1985].

Статья [Gray et al. 1996] является основополагающей работой, в которой заложена общая основа для сравнения различных алгоритмов репликации и высказана мысль, что энергичная репликация влечет за собой проблемы (это положило начало потоку работ по ленивым методам). Именно на ней основана классификация, которой мы пользовались в этой главе. Более детальная классификация приведена в работе [Wiesmann et al. 2000].

Определение согласованности в конечном счете взято из работы [Saito and Shapiro 2005], эпсилон-сериализуемость определена в статье [Pu and Leff 1991], а затем обсуждалась в работах [Ramamritham and Pu 1995] и [Wu et al. 1997]. Недавний обзор оптимистических (или ленивых) методов репликации см. в работе [Saito and Shapiro 2005]. В целом эта тема подробно обсуждается в работе [Kemme et al. 2010].

Актуальность, особенно для ленивых методов, стала предметом ряда исследований. Различные методы обеспечения «лучшей» актуальности обсуждаются в работах [Pacitti et al. 1998, 1999, Pacitti and Simon 2000, Röhm et al. 2002, Pape et al. 2004, Akal et al. 2005, Bernstein et al. 2006].

Обобщение изоляции моментальных снимков на реплицированные базы данных предложено в работе [Lin et al. 2005], а его использование в реплицированных базах данных обсуждается в работах [Plattner and Alonso 2004, Daudjee and Salem 2006]. RC-сериализуемость как еще одна слабая форма сериализуемости впервые определена в работе [Bernstein et al. 2006]. Сильная сериализуемость как в одной копии обсуждается в работе [Daudjee and Salem 2004], а сильная изоляция снимков – в работе [Daudjee and Salem 2006]; эти методы предотвращают инверсию транзакций.

Один из первых энергичных протоколов репликации с ведущей копией был реализован в распределенной версии INGRES и описан в работе [Stonebraker and Neuhold 1977].

Что касается ленивого подхода к репликации с единственной главной копией, то использовать граф репликации для упорядочения транзакций актуализации было предложено в статье [Breitbart and Korth 1997]. Идея обработки отложенных обновлений путем поиска подходящего назначения ведущих узлов для элементов данных высказана в работе [Chundi et al. 1996].

В работе [Bernstein et al. 2006] предложен ленивый алгоритм репликации с полной прозрачностью.

Использование групповой коммуникации обсуждалось в работах [Chockler et al. 2001, Stanoi et al. 1998, Kemme and Alonso 2000a,b, Patiño-Martínez et al. 2000, Jiménez-Peris et al. 2002]. Энергичный распределенный протокол, рассмотренный нами в разделе 6.4, взят из работы [Kemme and Alonso 2000b], а ленивый централизованный протокол – из работы [Pacitti et al. 1999].

Протокол доступных копий из раздела 6.5.2 описан в работах [Bernstein and Goodman 1984] и [Bernstein et al. 1987].

Существует много разных протоколов на основе кворума. Некоторые из них обсуждаются в работах [Triantafillou and Taylor 1995, Paris 1986, Tanenbaum and van Renesse 1988]. Оригинальный протокол голосования предложен в работе [Thomas 1979], а первое предложение по использованию голосования на основе кворума для управления репликами – в работе [Gifford 1979]. Представленный в разделе 6.5.2 алгоритм, который решает проблемы производительности, присущие алгоритму Гиффорда, взят из работы [El Abbadi et al. 1985]. Исчерпывающее исследование, на которое мы ссылаемся в том же разделе и которое свидетельствует о преимуществах семейства протоколов ROWA-A, – работа [Jiménez-Peris et al. 2003]. Помимо описанных нами алгоритмов, заслуживают внимания представленные в работах [Davidson 1984, Eager and Sevcik 1983, Herlihy 1987, Minoura and Wiederhold 1982, Skeen and Wright 1984, Wright 1983]. Эти алгоритмы сообща называются статическими, поскольку назначение голосов и определение кворумов чтения и записи задается априори. Анализ одного такого протокола (вообще анализ встречается редко) приведен в работе [Kumar and Segev 1993]. Примеры динамических протоколов репликации имеются в работах [Jajodia and Mutchler 1987, Barbara et al. 1986, 1989] и ряде других. Можно также изменять способ репликации данных. Такие протоколы называются адаптивными, за примером обратитесь к работе [Wolfson 1987].

Интересный алгоритм репликации, основанный на экономических моделях, описан в работе [Sidell et al. 1996].

Упражнения

Задача 6.1. Для каждого из четырех протоколов репликации (энергичный централизованный, энергичный распределенный, ленивый централизованный, ленивый распределенный) приведите сценарий или приложение, в котором этот подход лучше остальных. Обоснуйте свой ответ.

Задача 6.2. Компания располагает четырьмя территориально разнесенными складами для хранения и торговли. Рассмотрим следующую частичную схему базы данных:

```
ITEM(ID, ItemName, Price, …)
STOCK(ID, Warehouse, Quantity, …)
CUSTOMER(ID, CustName, Address, CreditAmt, …)
CLIENT-ORDER(ID, Warehouse, Balance, …)
ORDER(ID, Warehouse, CustID, Date)
ORDER-LINE(ID, ItemID, Amount, …)
```

База данных содержит отношения с информацией о товарах (ITEM содержит общую информацию, STOCK – количество каждого товара на каждом складе). Кроме того, в базе хранится информация о клиентах (заказчиках), например общая информация в таблице CUSTOMER. Основные операции, связанные с заказчиками, – заказ товаров, оплата счетов и общие запросы. Информация о заказах хранится в нескольких таблицах. Каждый заказ регистрируется в таблицах ORDER и ORDER-LINE. Каждый заказ представлен одной записью в таблице ORDER, содержащей следующие поля: идентификатор заказа, идентификатор заказчика, номер склада, на который поступил заказ, дата заказа и др. У клиента может быть несколько заказов, ожидающих исполнения на одном складе. Один заказ может включать несколько товаров. В таблице ORDER-LINE каждому товару в одном заказе соответствует одна строка. CLIENT-ORDER – сводная таблица, в которой каждой паре (клиент, склад) соответствует одна строка, содержащая суммарную стоимость всех заказов данного клиента с данного склада.

a) В компании имеется группа обслуживания клиентов, включающая несколько сотрудников, которые получают информацию о заказах и платежах клиентов, запрашивают данные о локальных клиентах для выписки счетов или регистрации платежных чеков и т. д. Кроме того, они отвечают на все вопросы заказчиков. Например, при заказе товаров изменяются (производится обновление и вставка) таблицы CLIENT-ORDER, ORDER, ORDER-LINE и STOCK. Для большей гибкости каждый сотрудник должен иметь возможность работать с любыми клиентами. Оценочно в рабочей нагрузке 80 % составляют запросы и 20 % обновления. Поскольку рабочая нагрузка смещена в сторону запросов, руководство приняло решение создать кластер персональных компьютеров, на каждом из которых хранится собственная база данных, с целью ускорить запросы за счет локального доступа. Как бы вы организовали репликацию данных, чтобы добиться этой цели? Какой протокол (или протоколы) управления репликами вы бы выбрали для поддержания согласованности данных?

b) Ежеквартально руководство компании должно принимать решение об ассортименте и стратегии продаж. Для этого необходимо непрерывно отслеживать и анализировать продажи различных товаров с разных складов, а также наблюдать за поведением потребителей. Как бы вы организовали репликацию данных для решения этой задачи? Какой протокол (или протоколы) управления репликами вы бы выбрали для поддержания согласованности данных?

Задача 6.3 (*). Альтернативный подход, позволяющий гарантировать, что в ленивых протоколах с единственным главным узлом и ограниченной прозрачностью транзакции актуализации можно применить во всех подчиненных узлах в одном и том же порядке, – использование графа репликации, описанного в разделе 6.3.3. Разработайте метод распределенного управления графом репликации.

Задача 6.4. Рассмотрим элементы данных *x* и *y*, реплицированные между узлами следующим образом:

Узел 1	Узел 2	Узел 3	Узел 4
x	*x*		*x*
	y	*y*	*y*

a) Назначьте голоса каждому узлу и определите кворумы чтения и записи.

b) Установите все возможные способы разделения сети и для каждого из них определите, в какой группе узлов можно завершать транзакцию, которая обновляет (читает и записывает) *x*, и каким должно быть условие завершения.

c) Решите задачу (b) для *y*.

Глава 7

Интеграция баз данных – системы управления мультибазами данных

До сих пор мы рассматривали распределенные СУБД, спроектированные сверху вниз. В частности, в главе 2 обсуждались методы секционирования и размещения базы данных, а в главе 4 – распределенная обработка запросов к такой базе данных. Эти методы и подходы годятся для тесно интегрированных однородных распределенных баз данных. В этой главе мы займемся распределенными базами данных, спроектированными снизу вверх, – в главе 1 мы назвали их системами управления мультибазами данных. В этом случае уже существует ряд баз данных, а задача проектировщика – интегрировать их в одну. В начале восходящего проектирования имеется набор локальных концептуальных схем (ЛКС). Процесс заключается в объединении локальных баз данных с их (локальными) схемами в глобальную базу данных и порождении глобальной концептуальной схемы (ГКС), которая также называется *опосредованной схемой*. Запрос к системе мультибаз данных более сложен, поскольку приложения и пользователи могут опрашивать систему либо через ГКС (или определенные над ней представления), либо через ЛКС, т. к. для работы с каждой локальной базой данных уже могут быть написаны приложения. Поэтому в методы обработки запросов, описанные в главе 4, придется внести корректировки, хотя многие идеи переносятся на мультибазы.

Интеграция баз данных и связанная с этим проблема опроса мультибаз – лишь одна сторона более общей проблемы *интероперабельности*, которая включает источники данных, не являющиеся базами данных, и интероперабельность на уровне приложений, а не только баз данных. Мы разобьем обсуждение на три части: в этой главе сосредоточимся на интеграции баз данных и проблеме запросов, в главе 12 обсудим вопросы, связанные с интеграцией веб-данных и доступом к ним, в главе 10 – более общую проблему интеграции данных из произвольных источников – *озера данных*.

Эта глава состоит из двух разделов. В разделе 7.1 мы обсудим интеграцию баз данных – процесс проектирования снизу вверх, а в разделе 7.2 – запросы к таким системам.

7.1. Интеграция баз данных

Интеграция баз данных может быть физической или логической. В первом случае исходные базы данных интегрируются, и интегрированная база данных *материализуется*. Такие образования называются *хранилищами данных* (data warehouses). Интеграция осуществляется с помощью средств «*извлечения, преобразования, загрузки*» (extract–transform–load – ETL), которые позволяют извлекать данные из различных источников, преобразовывать их в соответствии с ГКС и загружать (т. е. материализовывать). Этот процесс изображен на рис. 7.1. В случае логической интеграции глобальная концептуальная (или опосредованная) схема целиком *виртуальна* и не материализуется.

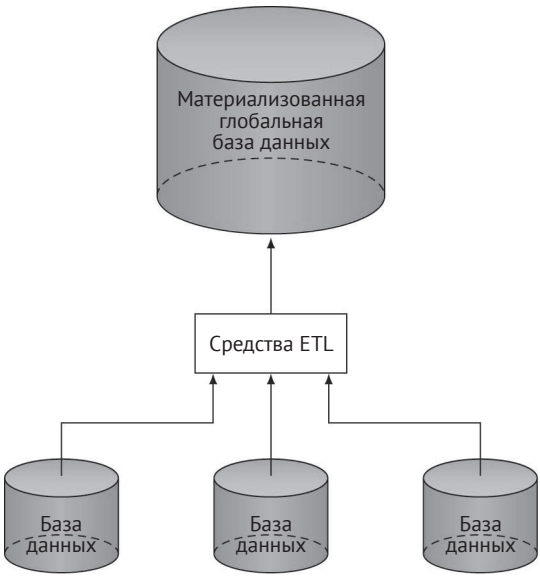


Рис. 7.1 ❖ Хранилище данных

Эти два подхода взаимно дополняют друг друга и предназначены для решения разных задач. Хранилища данных применяются в приложениях для поддержки принятия решений, которые часто называют *оперативной аналитической обработкой* (Online Analytical Processing – OLAP). Напомним (см. главу 5), что OLAP-приложения анализируют исторические обобщенные данные, поступающие из различных операционных баз данных, с помощью сложных запросов к потенциально очень большим таблицам. Следовательно, в хранилищах данных собираются и материализуются данные из многих операционных баз. Обновления, произведенные в операционных базах, переносятся в хранилище данных, эта процедура называется *обслуживанием материализованных представлений*.

С другой стороны, в случае логической интеграции данных интеграция является чисто виртуальной, и никакой материализованной глобальной

базы данных не существует (см. рис. 1.13). Данные находятся в операционных базах, а ГКС предоставляет виртуальную интеграцию для запросов к нескольким базам. В таких системах ГКС может быть либо определена заранее, и тогда локальные базы данных (т. е. ЛКС) отображаются на нее, либо она определяется снизу вверх путем объединения частей ЛКС локальных баз. Поэтому ГКС может не охватывать всю информацию, представленную в каждой ЛКС. Пользовательские запросы предъявляются к этой глобальной схеме, а затем подвергаются декомпозиции и направляются локальным операционным базам данных для обработки, как в тесно интегрированных системах, – основное отличие заключается в автономности и потенциальной гетерогенности локальных систем. Это имеет важные последствия для обработки запросов – темы раздела 7.2. Несмотря на обилие работ по управлению транзакциями в таких системах, поддержка глобальных обновлений весьма трудна вследствие автономности составляющих операционных СУБД. Поэтому они в основном используются только для чтения.

Для обозначения логической интеграции данных и получающихся в результате систем употребляются разные термины; наиболее распространенными, пожалуй, являются *интеграция данных* и *интеграция информации*, хотя чаще под этим понимают нечто большее, чем просто интеграция баз данных и включение данных из разных источников. В этой главе мы сосредоточимся на интеграции автономных и (возможно) гетерогенных баз данных, поэтому будем использовать термин *интеграция баз данных*, или *системы управления мультибазами данных* (СУМБД).

7.1.1. Методология проектирования снизу вверх

Проектирование снизу вверх (или восходящее проектирование) – это процесс объединения (физического или логического) составляющих баз данных в единую глобальную базу. Как уже отмечалось, иногда глобальную концептуальную (или опосредованную) схему определяют заранее, и тогда проектирование снизу вверх сводится к отображению всех ЛКС на эту схему. А бывает, что ГКС определяется как результат объединения частей ЛКС. В этом случае в процессе восходящего проектирования нужно сгенерировать ГКС и отобразить на нее отдельные ЛКС.

Если ГКС определена заранее, то между ГКС и ЛКС возможна одна из двух связей: «локальная как представление» (ЛКП) или «глобальная как представление» (ГКП). В первом случае определение ГКС существует, и каждая ЛКС рассматривается как представление над ней. Во втором случае ГКС определяется как набор представлений над ЛКС. Эти представления показывают, как можно при необходимости получить элементы ГКС из элементов ЛКС. Различие между двумя способами можно интерпретировать в терминах получаемых результатов. В ГКП результаты запросов ограничены наборами объектов, определенных в ГКС, хотя локальные СУБД могут быть значительно богаче (рис. 7.2a). С другой стороны, в ЛКП результаты ограничены объектами в локальных СУБД, хотя определение ГКС может быть богаче (рис. 7.2b). Таким образом, в ЛКП-системах, возможно, придется иметь дело с неполны-

ми ответами. Предлагался также гибридный подход – «глобальная и локальная как представление» (ГЛКП), когда связь между ГКС и ЛКС описывается с помощью ЛКП и ГКП одновременно.

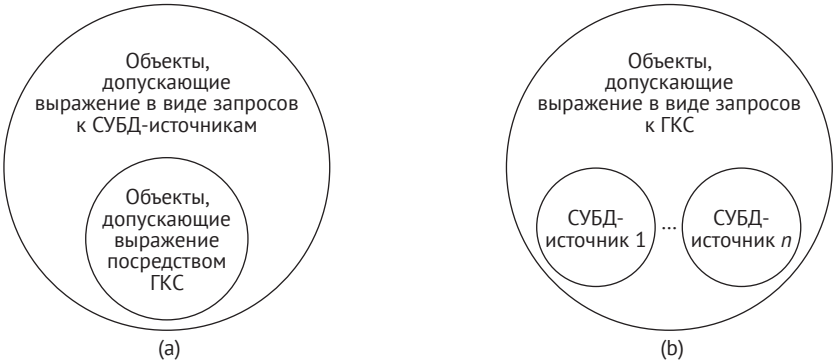


Рис. 7.2 ❖ ЛКП и ГКП (взято из работы [Koch 2001])

В процессе проектирования снизу вверх можно выделить два общих шага (рис. 7.3): *трансляция схемы* (или просто *трансляция*) и *генерирование схемы*. На первом шаге схемы баз данных транслируются в общее промежуточное каноническое представление (InS_1, InS_2, ..., InS_n). Использование канонического представления упрощает процесс трансляции за счет уменьшения количества трансляторов, которые нужно написать. Выбор канонической модели очень важен. Вообще говоря, она должна быть достаточно выразительной, чтобы вместить концепции, присутствующие во всех базах данных, которые потом предстоит интегрировать. Из предлагавшихся вариантов отметим модель сущность–связь, объектно-ориентированную модель и граф общего вида или его более простые формы: дерево или XML. В этой главе мы будем использовать в качестве канонической модели данных просто реляционную модель, несмотря на ее известные недостатки в плане представления развитых семантических концепций. Этот выбор не оказывает принципиального влияния на обсуждение главных вопросов интеграции данных. В любом случае мы не собираемся обсуждать особенности трансляции различных моделей данных в реляционную, эта тема освещается во многих учебниках по базам данных.

Очевидно, что шаг трансляции необходим, только если составляющие базы данных гетерогенны и локальные схемы определены с использованием разных моделей данных. Была проделана определенная работа по разработке федерации систем, когда системы с похожими моделями данных интегрируются вместе (например, реляционные системы интегрируются в одну концептуальную схему, а, скажем, объектные базы данных – в другую), а на следующем этапе эти интегрированные схемы «объединяются» (например, проект AURORA). В этом случае шаг трансляции откладывается, что дает приложениям более гибкие возможности доступа к исходным источникам данных наиболее подходящим для них способом.

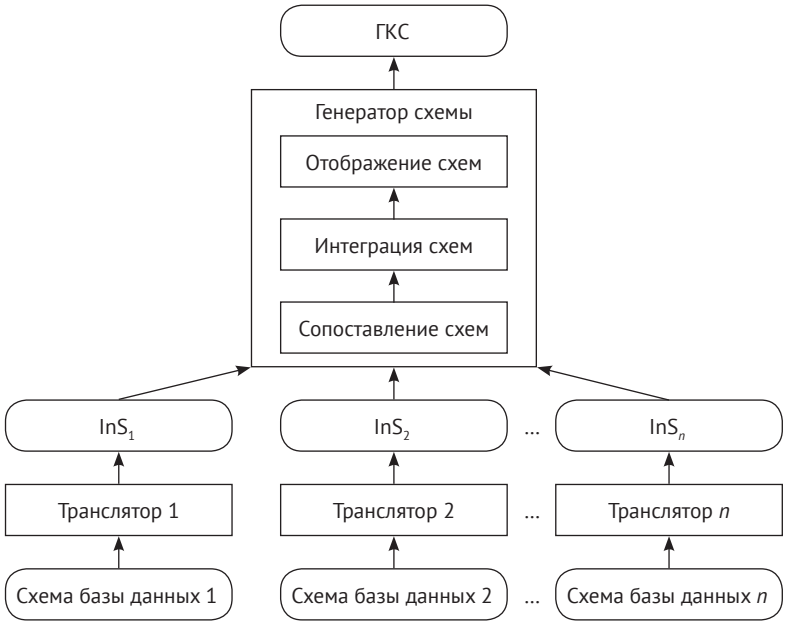


Рис. 7.3 ❖ Процесс интеграции баз данных

На втором шаге проектирования снизу вверх промежуточные схемы используются для генерирования ГКС. Процесс генерирования схемы состоит из следующих шагов:

1) сопоставление схем для определения синтаксических и семантических соответствий между транслированными элементами ЛКС или между элементами отдельных ЛКС и элементами предопределенной ГКС (раздел 7.1.2);
2) интеграция общих элементов схем в глобальную концептуальную (опосредованную) схему, если та еще не определена (раздел 7.1.3);
3) отображение схем, определяющее, как элементы каждой ЛКС отображаются на другие элементы ГКС (раздел 7.1.4).

Шаг отображения можно подразделить на два этапа: генерирование ограничений и генерирование преобразования. На первом этапе для заданного соответствия между двумя схемами генерируется функция преобразования, например запрос или определение представления над схемой источника, которая «заполнит» целевую схему. На втором этапе генерируется исполняемый код этой функции, который фактически создает целевую базу данных, согласованную с этими ограничениями. В некоторых случаях ограничения неявно включены в соответствия, что позволяет обойтись без первого этапа.

Пример 7.1. Чтобы было проще обсуждать проектирование глобальной схемы в мультибазовых системах, возьмем пример, расширяющий демонстрационную базу данных, с которой мы работаем в этой книге. Для демонстрации обоих шагов процесса интеграции баз данных введем в пример гетерогенность.

Рассмотрим две организации, каждая со своим определением базы данных. Одна – реляционная – база была представлена в главе 2. На рис. 7.4 ее определение для удобства повторено. Во второй базе данных хранятся аналогичные данные, но для определения используется модель данных сущность-связь (E-R), как показано на рис. 7.5[1].

```
EMP(ENO, ENAME, TITLE)
PROJ(PNO, PNAME, BUDGET, LOC)
ASG(ENO, PNO, RESP, DUR)
PAY(TITLE, SAL)
```

Рис. 7.4 ❖ Реляционное представление базы данных конструкторской компании

Предполагается, что читатель знаком с моделью данных сущность–связь. Поэтому мы не станем описывать сам формализм, а сделаем лишь несколько замечаний относительно семантики рис. 7.5. Эта база данных похожа на определение реляционной базы данных конструкторской компании на рис. 7.4 с одним важным отличием: в ней дополнительно хранятся данные

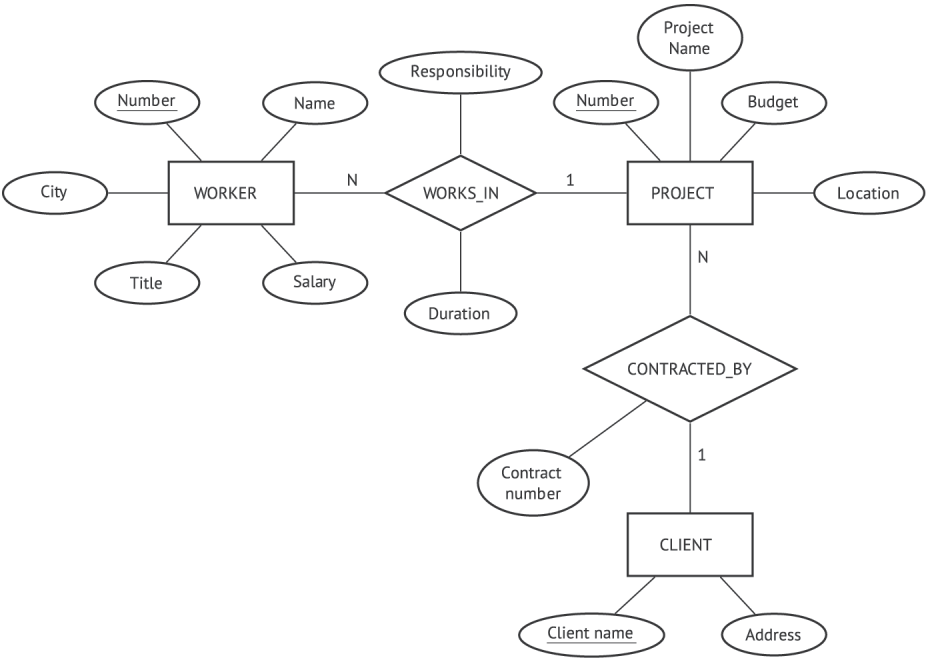


Рис. 7.5 ❖ База данных, представленная моделью сущность–связь

[1] В этой главе мы по-прежнему записываем имена отношений моноширинным шрифтом, но для записи компонент модели E-R будем использовать обычный шрифт, чтобы было проще различить их.

о заказчиках, для которых выполняются проекты. Прямоугольниками на рис. 7.5 представлены моделируемые сущности, а ромбами – связи между сущностями. Тип связи показан на линиях, исходящих из ромба. Например, CONTRACTED-BY – связь типа многие-к-одному между сущностями PROJECT и CLIENT (у одного проекта может быть только один заказчик, но у каждого заказчика может быть много проектов). Аналогично, WORKS-IN – связь типа многие-ко-многим между сущностями, которые она соединяет. Атрибуты сущностей и связей показаны в виде эллипсов. ♦

Пример 7.2. Отображение модели сущность–связь на реляционную модель показано на рис. 7.6. Мы переименовали некоторые атрибуты, чтобы имена были уникальными. ♦

```
WORKER(WNUMBER, NAME, TITLE, SALARY,CITY)
PROJECT(PNUMBER, PNAME, BUDGET)
CLIENT(CNAME, ADDRESS)
WORKS_IN(WNUMBER, PNUMBER, RESPONSIBILITY, DURATION)
CONTRACTED_BY(PNUMBER, CNAME, CONTRACTNO)
```

Рис. 7.6 ❖ Отображение ER-схемы на реляционную схему

7.1.2. Сопоставление схем

Если имеются две схемы, то процедура сопоставления для каждой концепции в одной схеме определяет соответствующую ей концепцию в другой. Как уже было сказано, если ГКС уже определена, то одной из схем обычно является ГКС, а задача в том, чтобы сопоставить каждую ЛКС с ГКС. В противном случае сопоставляются две ЛКС. Результаты сопоставления затем используются при отображении схем для порождения множества направленных отображений, которые, будучи применены к исходной схеме, отображают ее концепции в целевую.

Отображения, которые определены или выявлены в процессе сопоставления схем, задаются в виде набора правил, в котором каждое правило (*r*) определяет *соответствие* (*c*) между двумя элементами, *предикат* (*p*), который говорит, когда соответствие может иметь место, и *степень сходства* (*s*) между двумя соответственными элементами. Соответствие может просто идентифицировать две похожие концепции (мы будем обозначать это ≈) или может являться функцией, описывающей, как с помощью вычисления вывести одну концепцию из другой (например, если величина бюджета одного проекта выражена в долларах США, а другого – в евро, то соответствие может заключаться в том, что одна величина получается из другой умножением на обменный курс). Предикат – это условие, которое уточняет соответствие, описывая, когда оно может применяться. Например, в примере с бюджетом предикат *p* может специфицировать, что данное правило применяется, если один проект исполняется в США, а другой – в зоне евро. Степень сходства для каждого правила может задаваться или вычисляться. Это вещественное

число в диапазоне [0,1]. Таким образом, мы можем определить множество соответствий $M = \{r\}$, где $r = \langle c, p, s \rangle$.

Выше было отмечено, что соответствия могут быть определены или выявлены. При всем желании автоматизировать этот процесс он осложняется многими факторами. Самый важный из них – гетерогенность схем, т. е. различия между тем, как реальные явления отражаются в разных схемах. Этот вопрос настолько важен, что мы посвятим ему отдельный раздел 7.1.2.1. Помимо гетерогенности схем, есть и другие моменты, затрудняющие процесс сопоставления.

- *Недостаточная информация в схеме и данных*. Алгоритмы сопоставления опираются на информацию, которую можно извлечь из схемы и существующих примеров данных. Иногда ее недостаточность приводит к неоднозначности. Например, использование коротких имен концепций или неоднозначных аббревиатур, как в наших примерах, может стать причиной неправильного сопоставления.
- *Отсутствие документации по схеме*. В большинстве случаев схемы баз данных документированы плохо или не документированы вовсе. Очень часто человека, проектировавшего схему, уже нет, и проконсультироваться не с кем. Отсутствие этих важных источников информации дополнительно затрудняет сопоставление.
- *Субъективность сопоставления*. Наконец, важно понимать, что сопоставление элементов схемы – вещь очень субъективная; два проектировщика могут не прийти к согласию о «единственно правильном» сопоставлении. Поэтому оценить точность конкретного алгоритма очень сложно.

Тем не менее разработаны алгоритмические подходы к задаче сопоставления, и мы обсудим их в этом разделе. На алгоритм сопоставления влияет ряд факторов. Перечислим самые важные.

- *Сопоставление схем или примеров данных*. До сих пор в этой главе мы говорили об интеграции схем, поэтому естественно, что в фокусе внимания было сопоставление концепций двух схем между собой. Существует много алгоритмов, работающих с элементами схем. Но есть и другие, которые ориентированы на примеры данных или комбинацию схемы и данных. Идея в том, что рассмотрение примеров данных может разрешить некоторые семантические проблемы. Например, если имя атрибута неоднозначно, как, скажем, «contact-info», то выборка его значений может прояснить семантику: если данные записаны в формате номера телефона, то, очевидно, это номер телефона контактного лица, тогда как длинная строка букв может означать, что это имя контактного лица. Существует еще много атрибутов, например почтовые коды, названия стран, адреса электронной почты, которые легко можно опознать, анализируя примеры данных.

  Сопоставление, опирающееся только на информацию в схеме, может оказаться более эффективным, потому что не требует просмотра примеров данных. Кроме того, этот подход единственно возможный, если в сопоставляемых базах мало примеров, из-за чего обучение нельзя считать надежным. Но в некоторых случаях, например в одноранговых

системах (см. главу 9), схемы может вообще не быть, и тогда никакой альтернативы сопоставлению на основе анализа примеров попросту не существует.

- *На уровне элемента или на уровне структуры.* Некоторые алгоритмы сопоставления оперируют отдельными элементами схемы, тогда как в других принимаются во внимание также структурные связи между элементами. Основная идея подхода на уровне элементов – то, что значительная часть семантики схемы отражена в именах элементов. Но это не всегда помогает найти сложные соответствия, охватывающие несколько атрибутов. Алгоритмы сопоставления, учитывающие также структуру, исходят из того, что, как правило, структуры сопоставимых схем похожи.
- *Степень отображения.* Алгоритмы сопоставления обладают разными свойствами в части степени отображения. Простейший случай – отображение 1:1, когда каждому элементу одной схемы соответствует ровно один элемент другой схемы. Большинство алгоритмов относятся именно к этой категории, поскольку все проблемы в этом случае заметно упрощаются. Конечно, это предположение зачастую оказывается неверным. Например, атрибут «Total price» (Полная стоимость) можно отобразить на сумму двух атрибутов в другой схеме: «Subtotal» (Подытог) и «Taxes» (Налоги). Для таких отображений нужны более сложные алгоритмы сопоставления, учитывающие отображения вида 1:M и N:M.

Эти и другие критерии можно положить в основу классификации подходов к сопоставлению. Согласно данной классификации (которой мы будем придерживаться в этой главе с некоторыми модификациями), первый уровень разделения – алгоритмы на основе схем и на основе примеров данных (рис. 7.7). Алгоритмы на основе схем можно далее подразделить на алгоритмы на уровне элементов и на уровне структуры, тогда для алгоритмов на основе примеров имеет смысл только уровень структуры. На самом нижнем уровне проходит раздел между лингвистическими методами и методами на основе ограничений. Именно на этом уровне проявляются фундаментальные различия между алгоритмами сопоставления. Мы будем обсуждать лингвистические подходы в разделе 7.1.2.2, основанные на ограничениях в разделе 7.1.2.3, а основанные на обучении – в разделе 7.1.2.4. Каждый из этих алгоритмов называется *индивидуальным сопоставителем*, но возможны и их комбинации – *гибридные*, или *составные*, *сопоставители* (раздел 7.1.2.5).

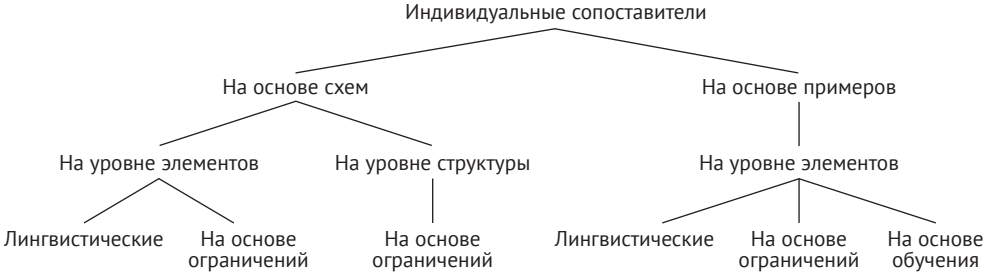


Рис. 7.7 ❖ Классификация методов сопоставления схем

7.1.2.1. Гетерогенность схем

Алгоритмы сопоставления схем имеют дело со структурной и семантической гетерогенностью схем. Мы обсудим этот вопрос сейчас, прежде чем описывать различные алгоритмы сопоставления.

Структурные конфликты бывают четырех типов: *конфликты типов*, *конфликты зависимостей*, *конфликты ключей* и *конфликты поведения*. Конфликт типов возникает, когда один и тот же объект представлен атрибутом в одной схеме и сущностью (отношением) в другой. Конфликт зависимостей возникает, когда для представления одного и того же в разных схемах используются связи разной степени (например, один-ко-многим и многие-ко-многим). Конфликт ключей возникает, когда есть несколько потенциальных ключей и в разных схемах выбраны разные первичные ключи. Конфликты поведения вызваны механизмом моделирования. Например, удаление последнего элемента в базе данных может приводить к удалению содержащей его сущности (например, после удаления последнего работника отдел расформировывается).

Пример 7.3. В сквозном примере из этой главы есть два структурных конфликта. Первый – конфликт типов для заказчиков проектов. В схеме на рис. 7.5 заказчик моделируется как сущность, а в схеме на рис. 7.4 – как атрибут отношения `PROJ`.

Второй структурный конфликт – конфликт зависимостей с участием связи WORKS_IN на рис. 7.5 и отношения `ASG` на рис. 7.4. В первом случае имеется связь типа многие-к-одному между WORKER и PROJECT, а во втором – связь типа многие-ко-многим. ♦

Структурные различия между схемами важны, но выявить и разрешить их недостаточно. При сопоставлении схем нужно также принимать во внимание семантику (возможно, разную) концепций схемы. Это называется *семантической гетерогенностью*. Данный термин сильно перегружен и нуждается в четком определении. Под ним понимают различия между базами данных, связанные со смыслом, интерпретацией и предполагаемым использованием данных. Предпринимались попытки формализовать семантическую гетерогенность и установить связь между ней и структурной гетерогенностью, но мы примем неформальный подход, ограничившись обсуждением на интуитивном уровне. Ниже перечислены некоторые проблемы, с которыми приходится сталкиваться алгоритмам сопоставления.

- *Синонимы*, *омонимы*, *гиперонимы*. Синонимы – это различные термины, относящиеся к одной и той же концепции. В нашем примере отношение `PROJ` и сущность PROJECT относятся к одной концепции. Омонимы – это употребления одного и того же термина для обозначения разных понятий в разных контекстах. В нашем примере BUDGET может означать бюджет-брутто в одной базе данных и бюджет-нетто (после вычета издержек) в другой, поэтому сравнивать их затруднительно. Гипероним – это термин, более общий, чем похожее слово. В рассматриваемой нами базе данных таких примеров нет, но, скажем, концепция Vehicle (транспортное средство) является гиперонимом для концепции Car

(автомобиль) (кстати говоря, Car в этом случае – *гипоним* Vehicle). Эти проблемы можно решить с помощью *онтологий предметной области*, которые определяют организацию концепций и терминов в конкретной предметной области.

- *Другая онтология*. Даже если онтологии используются для решения проблем в одной предметной области, часто бывает нужно сопоставить схемы из разных предметных областей. В этом случае нужно обращать особое внимание на смысл терминов в онтологиях, поскольку он может сильно зависеть от предметной области. Например, атрибут LOAD (нагрузка) может означать меру сопротивления в электротехнике, но меру веса в механике.
- *Неточный выбор слов*. Схемы могут содержать неоднозначные имена. Например, атрибуты LOCATION (в ER-модели) и `LOC` (в реляционной модели) в нашей базе данных могут обозначать полный адрес или только его часть. Аналогично имя «contact-info» может означать, что атрибут содержит имя контактного лица или номер его телефона. Такого рода неоднозначности встречаются часто.

7.1.2.2. Подходы на основе лингвистического сопоставления

Как следует из самого названия, в подходах на основе лингвистического сопоставления используются имена элементов и другая текстовая информация (например, текстовые описания и аннотации в определениях схем) для установления соответствия между элементами. Во многих случаях дополнительно могут использоваться внешние источники, например тезаурусы.

Лингвистические методы можно применять в алгоритмах на основе схем и на основе примеров. В первом случае сходство устанавливается между элементами схемы, во втором – между элементами отдельных примеров данных. Чтобы сделать обсуждение предметным, будем рассматривать в основном лингвистические методы на основе схем, а методы на основе экземпляров упомянем лишь вкратце. Поэтому будем применять нотацию ⟨`SC1.element-1` ≈ `SC2.element-2`, *p*, *s*⟩, означающую, что элемент `element-1` в схеме `SC1` соответствует элементу `element-2` в схеме `SC2` со степенью сходства *s*, если предикат *p* является истинным.

Лингвистические сопоставители, работающие на уровне элементов схемы, обычно интересуются именами элементов и обрабатывают синонимы, омонимы и гиперонимы. Иногда в определении схемы встречаются аннотации (комментарии на естественном языке), которыми может воспользоваться лингвистический сопоставитель. В подходах на основе примеров лингвистические сопоставители уделяют больше внимания методам информационного поиска, в т. ч. частотам слов, ключевым термам и т. д. В таких случаях сопоставитель «выводит» сходство из этих показателей.

В лингвистических сопоставителях схем используется набор лингвистических (или терминологических) правил, которые можно подготовить вручную или «выявить» с помощью вспомогательных источников данных, например таких тезаурусов, как WordNet. В случае вручную составленных правил проектировщик должен также задать предикат *p* и степень сходства *s*. Для выяв-

ленных правил они могут либо задаваться экспертом после выявления, либо вычисляться с применением одного из обсуждаемых ниже методов.

В составленных вручную лингвистических правилах могут учитываться такие вещи, как заглавные буквы, аббревиатуры и связи между понятиями. В некоторых системах правила задаются для каждой схемы индивидуально (*внутрисхемные правила*) проектировщиком, а затем *межсхемные правила* выявляются алгоритмом сопоставления. Но чаще всего база правил содержит как внутрисхемные, так и межсхемные правила.

Пример 7.4. В реляционной базе данных из примера 7.2 можно было бы определить (чисто интуитивно) следующий набор правил, в котором RelDB обозначает реляционную схему, а ERDB – транслированную схему сущность-связь:

⟨uppercase names ≈ lower case names, *true,* 1.0)⟩
⟨uppercase names ≈ capitalized names, *true,* 1.0)⟩
⟨capitalized names ≈ lower case names, *true,* 1.0)⟩
⟨RelDB.ASG ≈ ERDB.WORKS_IN, *true,* 0.8⟩
. . .

Первые три правила носят общий характер и говорят, как обращаться с заглавными буквами, а четвертое описывает сходство между отношением ASG в базе RelDB и сущностью WORKS_IN в базе ERDB. Поскольку эти соответствия должны быть верны в любом контексте, предикат $p = true$. ♦

Как мы уже сказали, существуют способы определить сходство имен элементов автоматически. Например, в системе COMA используются следующие методы определения сходства двух имен элементов:

- определяются *аффиксы*, т. е. общие префиксы и суффиксы строк, задающих имена элементов;
- сравниваются *n-граммы* имен. *N*-граммой называется подстрока длины *n*, а сходство строк тем выше, чем больше у них общих *n*-грамм;
- вычисляется *редакционное расстояние* между двумя строками. Редакционным расстоянием (или расстоянием Левенштейна) называется количество односимвольных операций (вставка, удаление, замена), необходимых для превращения одной строки в другую;
- вычисляется *soundex-индекс* имен элементов. Он определяет фонетическое сходство имен. Для англоязычных слов soundex-индекс слова получается путем его сворачивания в букву и три цифры. Результат приблизительно определяет звучание слова. Важно, что если два слова звучат похоже, то у них будут близкие soundex-индексы.

Пример 7.5. Рассмотрим сопоставление атрибутов RESP и RESPONSIBILITY в двух наших схемах. Правила, определенные в примере 7.4, убирают различия в регистре букв, так что остаются строки RESP и RESPONSIBILITY. Вычислим их сходство в смысле редакционного расстояния и *n*-грамм.

Количество односимвольных операций, необходимых для превращения одной строки в другую, равно 10 (либо добавить символы «O», «N», «S», «I», «B», «I», «L», «I», «T», «Y» в строку «RESP», либо удалить те же самые символы

из строки «RESPONSIBILITY»). Следовательно, доля изменений равна 10/14, и, стало быть, редакционное расстояние и сходство между строками равно 1 – (10/14) = 4/14 = 0.29.

Для вычисления *n*-грамм сначала выберем значение *n*. В данном примере положим *n* = 3, т. е. будем искать триграммы. В строке «RESP» есть всего две триграммы: «RES» и «ESP». Аналогично в строке двенадцать триграмм: «RES», «ESP», «SPO», «PON», «ONS», «NSI», «SIB», «IBI», «BIP», «ILI», «LIT», «ITY». Из этих двенадцати триграмм две совпадают, поэтому триграммное сходство равно 2/12 = 0.17. ♦

Все рассмотренные в этом разделе примеры попадают в категорию соответствий типа 1:1 – мы сопоставляли один элемент одной схемы с одним элементом другой схемы. Выше мы говорили, что бывают также соответствия типа 1:N (например, улица, номер дома, город и страна в одной базе данных могут быть извлечены из единственного элемента Address в другой), N:1 (например, полную стоимость Total_price можно вычислить по элементам Subtotal и Taxes) или N:M (например, название книги Book_title и ее рейтинг Rating можно получить соединением двух таблиц, в одной из которых хранится информация о книге, а в другой – читательские оценки и отзывы). Сопоставители типа 1:1, 1:N и N:1 обычно применяются для сопоставления на уровне элементов, а для сопоставления типа N:M нужны сопоставители на уровне схемы, потому что в этом случае без информации о схеме не обойтись.

7.1.2.3. Сопоставление на основе ограничений

Определения схем почти всегда содержат семантическую информацию, налагающую ограничения на значения элементов в базе данных. Обычно это информация о типе и допустимых диапазонах, ограничения ключа и т. д. В случае методов на основе примеров существующие диапазоны значений можно вывести, как и некоторые другие закономерности, присутствующие в данных. Всем этим могут воспользоваться сопоставители.

Рассмотрим типы данных, которые сообщают много семантической информации. Эту информацию можно использовать для разрешения неоднозначностей и для более точного сопоставления. Например, согласно вычислениям в примере 7.5, имена `RESP` и RESPONSIBILITY не слишком похожи. Однако если тип данных в обоих случаях одинаков, то его можно использовать для увеличения меры сходства. Аналогично сравнение типов данных позволяет провести различие между элементами с высокой степенью лексического сходства. Например, для атрибута `ENO` на рис. 7.4 редакционное расстояние и *n*-граммное сходство с двумя атрибутами NUMBER на рис. 7.5 одинаково (естественно, потому что рассматриваются только имена атрибутов). В таком случае могут помочь типы данных – если тип данных `ENO` и номера работника (WORKER.NUMBER) целый, а тип данных номера проекта (PROJECT.NUMBER) – строка, то `ENO` с большей вероятностью сопоставляется с WORKER.NUMBER.

В подходах на основе структуры для определения сходства элементов схемы можно использовать структурное сходство самих схем. Если элементы схемы структурно похожи, то уверенность в том, что они представляют одну

и ту же концепцию, выше. Например, если имена двух элементов сильно различаются и нам не удалось установить их сходство с помощью сопоставителей уровня элементов, но при этом у них имеются одинаковые свойства (например, одни и те же атрибуты) с одинаковыми типами данных, то шансы, что они представляют одну и ту же концепцию, повышаются.

Для определения структурного сходства нужно проверять сходство «окрестности» двух рассматриваемых концепций. Для определения окрестности обычно используется представление схем в виде графа, в котором каждой концепции (отношению, сущности, атрибуту) соответствует вершина, а между двумя вершинами проведено ориентированное ребро, если соответствующие им концепции как-то связаны (например, вершина, представляющая отношение, соединена ребром с вершинами, представляющими его атрибуты, а вершина, представляющая атрибут внешнего ключа, соединена с вершиной, представляющей соответствующий первичный ключ). В этом случае окрестность концепции можно определить как множество вершин, достижимых из соответствующей ей вершины по пути заранее заданной длины, и задача сводится к проверке сходства подграфов в этой окрестности. Во многих алгоритмах рассматривается дерево с корнем в исследуемой концепции и вычисляется сходство концепций, представленных корневыми вершинами деревьев. Основная идея заключается в том, что если подграфы (поддеревья) похожи, то больше шансов, что похожи концепции, представленные их «корневыми» вершинами. Сходство подграфов обычно устанавливается с помощью обхода снизу вверх, начиная с листьев, сходство которых определяется сопоставлением элементов (например, сходство имен с точностью до синонимов или совместимость типов данных). Сходство двух поддеревьев устанавливается рекурсивно на основе сходства их вершин и определяется как доля сильно связанных листьев этих поддеревьев. Это определение основано на предположении, что листовые вершины несут больше информации и что структурное сходство двух нелистовых элементов схем определяется сходством листовых вершин в соответствующих поддеревьях, даже если их непосредственные потомки не похожи. Это всего лишь эвристические правила, можно определить и другие.

Еще один интересный подход к учету окрестностей в ориентированных графах при вычислении сходства вершин называется *распространением сходства* (similarity flooding). В начальном графе сходство вершин уже определено с помощью сопоставителя элементов, а затем итеративно распространяется, чтобы определить сходство каждой вершины с ее соседями. Таким образом, если два элемента в разных схемах признаны похожими, то сходство соседних с ними вершин увеличивается. Итеративный процесс завершается, когда значения сходства вершин стабилизируются. Чтобы уменьшить объем работы, на каждой итерации выбирается подмножество вершин, считающихся «наиболее вероятными» соответствиями, оно и рассматривается на следующей итерации.

В обоих подходах никак не учитывается семантика ребер. В некоторых графовых представлениях с ребрами связывается дополнительная семантика. Например, *ребра включения*, идущие от вершины, представляющей отношение или сущность, к вершинам атрибутов, могут отличаться от *ссылочных ребер*, идущих от вершины атрибута внешнего ключа к вершине атрибута со-

ответствующего первичного ключа. В некоторых системах (например, DIKE) такая семантика ребер используется.

7.1.2.4. Сопоставление на основе обучения

Третий из предложенных подходов к сопоставлению схем основан на использовании методов машинного обучения. В них сопоставление рассматривается как задача классификации, когда концепции из разных схем распределяются по классам согласно их сходству. Сходство определяется с помощью проверки признаков примеров данных из баз, соответствующих схемам. Чтобы понять, как следует классифицировать концепции по их признакам, производится анализ примеров данных из обучающего набора.

Процесс устроен следующим образом (рис. 7.8). Подготавливается обучающий набор (τ), который содержит примеры соответствий между концепциями из двух баз данных D_i и D_j. Этот набор можно сгенерировать после ручной идентификации соответствий между схемами обеих баз, за которой следует выборка обучающих примеров данных, или посредством задания запроса, преобразующего данные одной базы в данные другой. Обучаемый использует эти данные для получения вероятностной информации о признаках. Затем классификатор, которому предъявлены два других экземпляра баз данных (D_k и D_l), использует эту информацию, чтобы на основе анализа данных в D_k и D_l сделать предсказания о классификации элементов этих баз.

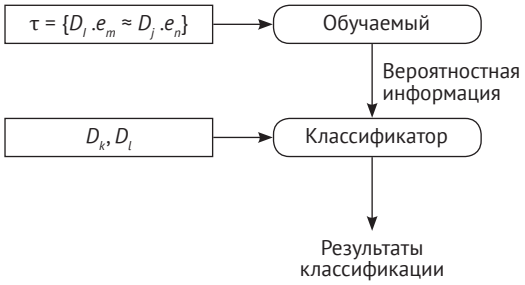


Рис. 7.8 ❖ Подход к сопоставлению на основе обучения

Этот общий подход применим ко всем алгоритмам сопоставления схем на основе обучения. Отличаются они только типом обучаемого и настройкой поведения обучаемого для сопоставления схем. В одних случаях используются нейронные сети (например, SEMINT), в других – наивный байесовский классификатор (Autoplex, LSD), в третьих – решающие деревья. Мы не будем обсуждать детали этих методов обучения.

7.1.2.5. Комбинированные подходы к сопоставлению

У каждого из рассмотренных методов индивидуального сопоставления есть плюсы и минусы. В конкретной ситуации одни оказываются лучше, другие хуже. Поэтому во «всеохватывающем» алгоритме или методике сопоставления обычно приходится использовать несколько индивидуальных сопоставителей.

Есть два способа объединения сопоставителей: гибридный и составной. В *гибридных* методиках несколько сопоставителей комбинируются в одном алгоритме. Иными словами, элементы из двух схем сравниваются с помощью нескольких сопоставителей элементов (например, сравнение строк и типов данных) и (или) сопоставителей структур, в результате чего определяется итоговое сходство. Внимательный читатель наверняка заметил, что при обсуждении основанных на ограничениях алгоритмов сопоставления, ориентированных на структурное сопоставление, мы применяли гибридный подход, потому что на начальной стадии устанавливалось соответствие листовых узлов с помощью сопоставителя элементов, а затем вычисленные значения использовались для структурного сопоставления. С другой стороны, в *составных* алгоритмах каждый сопоставитель применяется к элементам обеих схем (или двум примерам) по отдельности, так что получается набор оценок сходства, которые затем объединяются в комбинированную оценку. Точнее, если $s_i(C_j^k, C_l^m)$ – оценка сходства, выданная сопоставителем i ($i = 1, \ldots, q$) для концепций C_j из схемы k и C_l из схемы m, то составное сходство этих концепций равно $s(C_j^k, C_l^m) = f(s_1, \ldots, s_q)$, где f – функция объединения оценок сходства. Можно использовать как простые функции, например `average`, `max` или `min`, так и более сложные функции агрегирования и ранжирования, которые мы будем обсуждать в разделе 7.2. Составной подход применялся в системах LSD и iMAP для обработки сопоставлений типа 1:1 и N:M соответственно.

7.1.3. Интеграция схем

После того как схемы сопоставлены, мы имеем соответствия между различными ЛКС. Следующий шаг – создание ГКС – называется *интеграцией схем*. Как уже отмечалось, этот шаг необходим, только если ГКС еще не определена и сопоставление производилось для отдельных ЛКС. Если же ГКС была определена заранее, то на этапе сопоставления устанавливаются соответствия между ней и каждой ЛКС, так что этап интеграции не нужен. Если ГКС создается в результате интеграции ЛКС на основе выявленных соответствий, то на этапе интеграции важно выявить соответствия между ГКС и различными ЛКС. Для процесса интеграции разработаны вспомогательные инструменты, но без участия человека, очевидно, не обойтись.

Пример 7.6. Существует много способов интегрировать две ЛКС из обсуждаемого нами примера. На рис. 7.9 показана одна из возможных ГКС, являющаяся результатом интеграции схем. В оставшейся части этой главы мы будем использовать именно ее. ♦

Методы интеграции можно отнести к двум категориям – бинарные и n-арные – в зависимости от того, как локальные схемы обрабатываются на первом этапе (рис. 7.10). В методах бинарной интеграции за один раз рассматриваются две схемы. Это можно делать пошагово (лесенкой), как на рис. 7.11a, когда создаются промежуточные схемы для интеграции с последующими, или попарно, как на рис. 7.11b, когда интегрируются между собой две схемы, а созданная в результате промежуточная схема интегрируется с другими промежуточными.

EMP(E#, ENAME, TITLE, CITY)
PAY(TITLE, SAL)
PR(P#, PNAME, BUDGET, LOC)
CL(CNAME, ADDR, CT#, P#)
WORKS(E#, P#, RESP, DUR)

Рис. 7.9 ❖ Пример интегрированной ГКС
(EMP – работник, PR – проект, CL – заказчик)

Рис. 7.10 ❖ Классификация методик интеграции

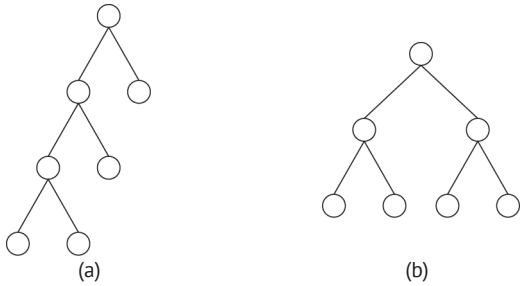


Рис. 7.11 ❖ Бинарные методы интеграции:
(a) пошаговый; (b) попарный

N-арные методы позволяют интегрировать более двух схем на каждой итерации. В случае однопроходной интеграции (рис. 7.12a) все схемы интегрируются сразу, и после одной итерации получается глобальная концептуальная схема. К достоинствам такого подхода следует отнести доступность полной информации обо всех базах данных на этапе интеграции. Никакой внутренне присущей очередности интеграции схем не существует. Различные компромиссы, например наилучшее представление элементов данных или самая понятная структура, возможны между всеми схемами, а не только их подмножеством. Но есть и недостатки – повышенная сложность и проблемы с автоматизацией.

Итеративная *n*-арная интеграция (рис. 7.12b) обладает большей гибкостью (как правило, доступно больше информации) и общностью (количество схем может варьироваться по желанию интегратора). Бинарные методы – частный случай *n*-арных итеративных. Они уменьшают потенциальную сложность интеграции и позволяют внедрить автоматизацию, поскольку количество схем, рассматриваемых на каждом шаге, проще контролировать. Интеграция с помощью *n*-арного процесса позволяет интегратору производить операции

с более чем двумя схемами. Из практических соображений в большинстве систем используется бинарная методика, но некоторые исследователи предпочитают *n*-арный подход за доступность полной информации.

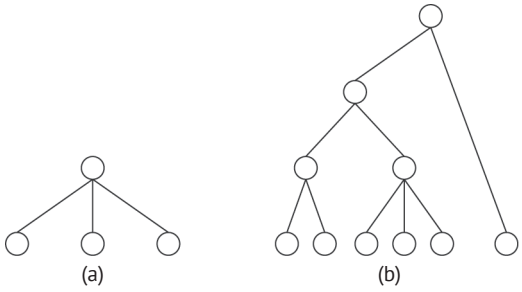


Рис. 7.12 ❖ *N*-арные методы интеграции:
(a) однопроходный; (b) итеративный

7.1.4. Отображение схем

После того как ГКС (или опосредованная схема) определена, необходимо решить, как на ГКС (цель) отображаются данные из каждой локальной базы (источника), сохранив при этом семантическую согласованность (определенную и в источниках, и в целевой схеме). Хотя при сопоставлении схем были выявлены соответствия между ЛКС и ГКС, возможно, не было явно определено, как получать глобальную базу данных из локальной. Именно для этого предназначен этап отображения схем.

В случае хранилищ данных отображения схем используются, чтобы явно извлекать данные из источников, преобразовывать их в схему хранилища и заполнять его. В случае систем интеграции данных эти отображения используются на этапе обработки запроса как процессором запросов, так и обертками (см. раздел 7.2).

Мы изучим два вопроса, связанных с отображением схем: *создание отображения* и *обслуживание отображения*. Под созданием отображения понимается процесс создания явных запросов, которые отображают данные из локальной базы в глобальную. Обслуживание отображения – это обнаружение и исправление несогласованностей отображения, появляющихся в результате эволюции схем. Исходные схемы могут претерпевать структурные или семантические изменения, из-за чего отображение перестает быть корректным. Задача обслуживания состоит в том, чтобы находить некорректные отображения и (автоматически) переписывать их, так чтобы восстановить семантическую согласованность с новой схемой, обеспечив при этом семантическую эквивалентность с текущим отображением.

7.1.4.1. Создание отображения

Создание отображения начинается с исходной ЛКС, целевой ГКС и набора соответствий между схемами *M*, а в результате порождается набор запросов,

после выполнения которых из исходных данных будут созданы данные, отвечающие ГКС. В хранилищах данных эти запросы действительно выполняются для создания хранилища (глобальной базы данных), тогда как в системах интеграции данных они используются в обратном направлении на этапе обработки запроса (раздел 7.2).

Добавим в обсуждение конкретики, обратившись к принятому нами каноническому реляционному представлению. Исходная ЛКС состоит из набора отношений *Source* = {O_1, …, O_m}, ГКС – из набора глобальных (или целевых) отношений *Target* = {T_1, …, T_n}, а M – это набор правил сопоставления схем, определенный в разделе 7.1.2. Мы ищем способ сгенерировать для каждого T_k запрос Q_k, определенный на подмножестве (возможно, собственном) отношений из *Source* и такой, что в результате его выполнения из исходных отношений порождаются данные T_k.

Это можно сделать итеративно, рассматривая каждое T_k по очереди. Мы начинаем с $M_k \subseteq M$ (M_k – множество правил, применимых только к атрибутам T_k) и разбиваем его на подмножества $\{M_k^1, \ldots, M_k^s\}$ такие, что каждое M_k^j задает один из возможных способов вычисления значений T_k. Каждое M_k^j можно отобразить в запрос q_k^j, при выполнении которого генерируются *некоторые* данные T_k. Объединение всех этих запросов дает искомый запрос $Q_k (= \cup_j q_k^j)$.

Алгоритм состоит из четырех шагов, которые мы обсудим ниже. В правилах не учитываются значения сходства. В обоснование этого решения можно сказать, что значения сходства будут использованы на последних стадиях процесса соответствия при окончательном формировании соответствий, поэтому на этапе отображения они не нужны. Кроме того, когда мы дошли до этого этапа процесса интеграции, нас интересует, как отобразить исходное отношение (ЛКС) в целевое (ГКС). Поэтому соответствия – это не симметричные эквиваленции ($\approx$), а отображения ($\mapsto$): атрибут(ы) (возможно, нескольких) исходных отношений в атрибут целевого отношения (т. е. ($O_i.attribute_k$, $O_j.attribute_l$) $\mapsto$ $T_w.attribute_z$).

Пример 7.7. Для демонстрации алгоритма мы возьмем другой пример базы данных, поскольку в нашем сквозном примере нет тех сложностей, которые мы хотим продемонстрировать. Новый пример будет абстрактным.

Исходные отношения (ЛКС):

$O_1(A_1, A_2)$;
$O_2(B_1, B_2, B_3)$;
$O_3(C_1, C_2, C_3)$;
$O_4(D_1, D_2)$.

Целевое отношение (ГКС):

$T(W_1, W_2, W_3, W_4)$.

Мы рассматриваем только одно отношение в ГКС, потому что алгоритм обходит целевые отношения по одному, так что для демонстрации его работы этого достаточно.

Связи внешнего ключа между атрибутами таковы:

Внешний ключ	Ссылается на
A_1	B_1
A_2	B_1
C_1	B_1

Предположим, что для атрибутов отношения T были найдены следующие соответствия (составляющие множество M_T). В последующих примерах предикаты нам будут не интересны, поэтому явно они не задаются.

$r_1 = \langle A_1 \mapsto W_1, p\rangle$;
$r_2 = \langle A_2 \mapsto W_2, p\rangle$;
$r_3 = \langle B_2 \mapsto W_4, p\rangle$;
$r_4 = \langle B_3 \mapsto W_3, p\rangle$;
$r_5 = \langle C_1 \mapsto W_1, p\rangle$;
$r_6 = \langle C_2 \mapsto W_2, p\rangle$;
$r_7 = \langle D_1 \mapsto W_4, p\rangle$. ♦

На первом шаге множество M_k (соответствующее T_k) разбивается на подмножества $\{M_k^1, \ldots, M_k^n\}$ такие, что каждое M_k^j содержит не более одного соответствия для каждого атрибута T_k. Эти подмножества называются *потенциальными множествами-кандидатами*; одни из них могут быть *полными*, т. е. включают соответствия для всех атрибутов, другие – нет. Есть две причины рассматривать неполные множества. Во-первых, может быть так, что для одного или нескольких атрибутов целевого отношения вообще не найдено соответствий (т. е. полных множеств просто не существует). Во-вторых, для большой и сложной схемы базы данных имеет смысл строить отображение итеративно, чтобы проектировщик мог задавать отображения инкрементно.

Пример 7.8. M_T разбито на 53 подмножества (потенциальные множества-кандидаты). Первые восемь из них полны, остальные нет. Некоторые из них приведены ниже. Чтобы было проще читать, полные правила перечислены в порядке целевых атрибутов, на которые производится отображение (т. е. третьим правилом в M_T^1 является r_4, потому что оно отображает на атрибут W_3):

$M_T^1 = \{r_1, r_2, r_4, r_3\}$;
$M_T^2 = \{r_1, r_2, r_4, r_7\}$;
$M_T^3 = \{r_1, r_6, r_4, r_3\}$;
$M_T^4 = \{r_1, r_6, r_4, r_7\}$;
$M_T^5 = \{r_5, r_2, r_4, r_3\}$;
$M_T^6 = \{r_5, r_2, r_4, r_7\}$;
$M_T^7 = \{r_5, r_6, r_4, r_3\}$;
$M_T^8 = \{r_5, r_6, r_4, r_7\}$;
$M_T^9 = \{r_1, r_2, r_3\}$;
$M_T^{10} = \{r_1, r_2, r_4\}$;
$M_T^{11} = \{r_1, r_3, r_4\}$;
$M_T^{12} = \{r_2, r_3, r_4\}$;
$M_T^{13} = \{r_1, r_3, r_6\}$;
$M_T^{14} = \{r_3, r_4, r_6\}$;
. . .
. . .
$M_T^{47} = \{r_1\}$;
$M_T^{48} = \{r_2\}$;

$M_{\mathsf{T}}^{49} = \{r_3\}$; $M_{\mathsf{T}}^{50} = \{r_4\}$;
$M_{\mathsf{T}}^{51} = \{r_5\}$; $M_{\mathsf{T}}^{52} = \{r_6\}$;
$M_{\mathsf{T}}^{53} = \{r_7\}$. ♦

На втором шаге алгоритм анализирует все потенциальные множества-кандидаты M_k^j, чтобы понять, можно ли для них сгенерировать «хороший» запрос. Если все соответствия в M_k^j отображают значения из одного исходного отношения в T_k, то сгенерировать запрос, соответствующий M_k^j, легко. Особый интерес представляют соответствия, которые требуют доступа к нескольким исходным отношениям. В этом случае алгоритм проверяет, существует ли ссылочная связь между этими отношениями посредством внешних ключей (т. е. существует ли путь соединения исходных отношений). Если нет, то это потенциальное множество-кандидат исключается из дальнейшего рассмотрения. Если путей соединения с помощью связей внешнего ключа несколько, то алгоритм ищет пути, порождающие наибольшее количество кортежей (т. е. такие, для которых оценка разности размеров внешнего и внутреннего соединений наименьшая). Если таких путей несколько, то выбрать один из них должен проектировщик базы данных (такие инструменты, как Clio, OntoBuilder и прочие, облегчают этот процесс и предоставляют средства для просмотра и задания соответствий). В результате этого шага получается набор $\overline{M_k} \subseteq M_k$ *множеств-кандидатов*.

Пример 7.9. В этом примере не существует множества M_k^j, в котором значения всех атрибутов T являются образами атрибутов одного исходного отношения. А из множеств, связанных с несколькими исходными отношениями, на «хорошие» запросы можно отобразить правила, в которых участвуют O_1, O_2 и O_3, потому что между ними имеются связи внешнего ключа. Однако правила, в которых участвует O_4 (т. е. те, что включают правило r_7), нельзя отобразить на «хороший» запрос, потому что не существует пути соединения O_4 с другими отношениями (любой запрос включал бы декартово произведение, а это дорого). Таким образом, эти правила исключаются из потенциальных множеств-кандидатов. Если рассматривать только полные множества, то M_k^2, M_k^4, M_k^6 и M_k^8 исключаются. В итоге набор множеств-кандидатов ($\overline{M_k}$) содержит 35 правил (читателям рекомендуется убедиться в этом, чтобы лучше понять алгоритм). ♦

На третьем шаге алгоритм ищет покрытие множеств-кандидатов $\overline{M_k}$. Покрытием $C_k \subseteq \overline{M_k}$ называется такой набор множеств-кандидатов, что каждое соответствие, встречающееся в $\overline{M_k}$, встречается хотя бы в одном C_k. Смысл нахождения покрытия заключается в том, что оно учитывает все соответствия, и, следовательно, его достаточно для генерирования целевого отношения T_k. Если покрытий несколько (одно соответствие может участвовать в нескольких покрытиях), то они ранжируются по возрастанию количества множеств-кандидатов. Чем меньше в покрытии множеств-кандидатов, тем меньше запросов придется генерировать на следующем шаге, что, в свою очередь, повышает эффективность сгенерированных отображений. Если имеется несколько покрытий одного ранга, то они дополнительно ранжируются по убыванию общего количества уникальных целевых атрибутов, использованных

в множествах-кандидатах, составляющих покрытие. Смысл этого действия в том, что покрытия с большим числом атрибутов порождают меньше null-значений в результате. На этой стадии, возможно, придется попросить проектировщика выбрать одно из ранжированных покрытий.

Пример 7.10. Сразу отметим, что рассмотрению подлежат шесть правил, определяющих соответствия в $\overline{M}_k$, поскольку множества M^j_k, включающие правило r_7, были исключены. Количество возможных покрытий велико; для демонстрации алгоритма начнем с тех, что включают M^1_k:

$$C^1_\text{т} = \left\{\underbrace{\{r_1, r_2, r_4, r_3\}}_{M^1_\text{т}}, \underbrace{\{r_1, r_6, r_4, r_3\}}_{M^3_\text{т}}, \underbrace{\{r_2\}}_{M^{48}_\text{т}}\right\};$$

$$C^2_\text{т} = \left\{\underbrace{\{r_1, r_2, r_4, r_3\}}_{M^1_\text{т}}, \underbrace{\{r_5, r_2, r_4, r_3\}}_{M^5_\text{т}}, \underbrace{\{r_6\}}_{M^{50}_\text{т}}\right\};$$

$$C^3_\text{т} = \left\{\underbrace{\{r_1, r_2, r_4, r_3\}}_{M^1_\text{т}}, \underbrace{\{r_5, r_6, r_4, r_3\}}_{M^7_\text{т}}\right\};$$

$$C^4_\text{т} = \left\{\underbrace{\{r_1, r_2, r_4, r_3\}}_{M^1_\text{т}}, \underbrace{\{r_5, r_6, r_4\}}_{M^{12}_\text{т}}\right\};$$

$$C^5_\text{т} = \left\{\underbrace{\{r_1, r_2, r_4, r_3\}}_{M^1_\text{т}}, \underbrace{\{r_5, r_6, r_3\}}_{M^{19}_\text{т}}\right\};$$

$$C^6_\text{т} = \left\{\underbrace{\{r_1, r_2, r_4, r_3\}}_{M^1_\text{т}}, \underbrace{\{r_5, r_6\}}_{M^{32}_\text{т}}\right\}.$$

Видно, что покрытия состоят из двух или трех множеств-кандидатов. Поскольку алгоритм предпочитает покрытия, содержащие меньше множеств-кандидатов, оставляем только те, в которых два множества. Эти покрытия различаются количеством целевых атрибутов. Поскольку алгоритм предпочитает покрытия с наибольшим числом целевых атрибутов в каждом множестве-кандидате, будет выбрано покрытие $C^3_\text{т}$.

Заметим, что в силу двух эвристик, используемых в алгоритме, рассмотрению подлежат только покрытия, в которых участвуют $M^1_\text{т}$, $M^3_\text{т}$, $M^5_\text{т}$ и $M^7_\text{т}$. Можно определить аналогичные покрытия с участием $M^3_\text{т}$, $M^5_\text{т}$ и $M^7_\text{т}$; оставляем это читателю в качестве упражнения. Далее предполагается, что проектировщик предпочел покрытие $C^3_\text{т}$. ♦

На последнем шаге алгоритм строит запрос q^j_k для каждого множества-кандидата, вошедшего в покрытие, выбранное на предыдущем шаге. Объединение всех этих запросов (`UNION ALL`) даст окончательное отображение для отношения T_k в ГКС.

Запрос q^j_k строится следующим образом:

- фраза **`SELECT`** включает все соответствия (c) в каждом из правил (r^i_k), входящих в M^j_k;
- фраза **`FROM`** включает все исходные отношения, упомянутые в r^i_k и в путях соединения, найденных на шаге 2 алгоритма;
- фраза **`WHERE`** включает конъюнкцию всех предикатов (p) в r^i_k и всех предикатов в соединениях, найденных на шаге 2 алгоритма;

- если r_k^i содержит агрегатную функцию в c или в p, то добавляется фраза **GROUP BY** по атрибутам (или функциям от атрибутов), которые входят во фразу **SELECT**, но не входят в агрегат;
- если агрегат находится в соответствии c, то он добавляется в **SELECT**, иначе (если агрегат находится в предикате p) создается фраза **HAVING** с этим агрегатом.

Пример 7.11. Поскольку в примере 7.10 мы решили использовать для построения окончательного отображения покрытие C_T^3, то нужно сгенерировать два запроса: q_T^1 и q_T^7, соответствующих множествам M_T^1 и M_T^7. Для удобства восприятия снова перечислим правила:

$r_1 = \langle \mathtt{A}_1 \mapsto \mathtt{W}_1, p \rangle$;
$r_2 = \langle \mathtt{A}_2 \mapsto \mathtt{W}_2, p \rangle$;
$r_3 = \langle \mathtt{B}_2 \mapsto \mathtt{W}_4, p \rangle$;
$r_4 = \langle \mathtt{B}_3 \mapsto \mathtt{W}_3, p \rangle$;
$r_5 = \langle \mathtt{C}_1 \mapsto \mathtt{W}_1, p \rangle$;
$r_6 = \langle \mathtt{C}_2 \mapsto \mathtt{W}_2, p \rangle$.

Запросы выглядят следующим образом:

q_k^1: **SELECT** $\mathtt{A}_1, \mathtt{A}_2, \mathtt{B}_2, \mathtt{B}_3$
FROM $\mathtt{O}_1, \mathtt{O}_2$
WHERE p_1 **AND** $\mathtt{O}_1.\mathtt{A}_2 = \mathtt{O}_2.\mathtt{B}_1$

q_k^7: **SELECT** $\mathtt{B}_2, \mathtt{B}_3, \mathtt{C}_1, \mathtt{C}_2$
FROM $\mathtt{O}_2, \mathtt{O}_3$
WHERE p_3 **AND** p_4 **AND** p_5 **AND** p_6 **AND** $\mathtt{O}_3.c_1 = \mathtt{O}_2.\mathtt{B}_1$

Таким образом, окончательный запрос Q_k для целевого отношения T принимает вид q_k^1 **UNION ALL** q_k^7. ♦

Результатом этого алгоритма после его итеративного применения к каждому целевому отношению $\mathtt{T}_k$ является множество запросов $Q = \{Q_k\}$, выполнение которых порождает данные для отношений, вошедших в ГКС. Таким образом, алгоритм порождает ГКП-отображения между реляционными схемами – напомним, что ГКП определяет ГКС как представление над ЛКС, а это именно то, что делает множество отображающих запросов. Алгоритм учитывает семантику исходных схем, потому что при решении о том, какие запросы генерировать, рассматриваются связи внешнего ключа. Однако семантика целевой схемы не учитывается, поэтому не гарантируется, что кортежи, сгенерированные в результате выполнения отображающих запросов, удовлетворяют этой семантике. Когда ГКС получается интеграцией ЛКС, это не так важно, но если ГКС определена независимо от ЛКС, то могут возникнуть проблемы.

Алгоритм можно улучшить, приняв во внимание семантику не только исходных схем, но и целевой. Для этого нужно рассматривать межсхемные зависимости при генерации кортежей. Иными словами, необходимо порождать ГЛКП-отображения. По определению, ГЛКП-отображение – не просто запрос к исходным отношениям, а связь между запросом к исходным отношениям

(т. е. ЛКС) и запросом к целевым отношениям (т. е. ГКС). Уточним эту мысль. Рассмотрим сопоставление схем v, определяющее соответствие между атрибутом A отношения R из исходной ЛКС и атрибутом B отношения T из целевой ГКС (в обозначениях этого раздела $v = \langle R.A \approx T.B, p, s \rangle$). Тогда исходный запрос определяет, как получить R.A, а целевой – как получить T.B. Стало быть, ГЛКП-отображение – связь между этими двумя запросами.

Поставленной цели можно достичь, начав с исходной схемы, целевой схемы и M и «обнаружив» отображения, которые удовлетворяют семантике исходной и целевой схем одновременно. Это более мощный алгоритм, чем тот, что мы обсуждали выше, поскольку он способен обрабатывать вложенные структуры, типичные для XML, объектных баз данных и реляционных систем, поддерживающих вложенность.

Первый шаг обнаружения всех отображений, основанных на соответствиях, выявленных при сопоставлении схем, – *семантическая трансляция*, цель которой – интерпретировать соответствия в M способом, согласованным с семантикой исходной и целевой схем, которая отражена в структуре схемы и ссылочных ограничениях (внешних ключах). Результатом является множество *логических отображений*, каждое из которых отражает проектные решения (семантику), принятые в исходной и целевой схемах. Каждое логическое отображение соответствует одному отношению в целевой схеме. Второй шаг – *трансляция данных*, когда каждое логическое отображение реализуется в виде правила, транслируемого в запрос, при выполнении которого будет создан экземпляр целевого элемента.

На вход семантической трансляции подается исходная схема *Source*, целевая схема *Target* и M. Выполняются следующие два шага:

- проверить внутрисхемную семантику *Source* и *Target* по отдельности и для каждой сформировать множество семантически согласованных *логических отношений*;
- интерпретировать межсхемные соответствия в M в контексте логических отношений, сгенерированных на шаге 1, и сформировать множество запросов Q, семантически согласованных с *Target*.

7.1.4.2. Обслуживание отображений

В динамических средах, где схемы эволюционируют со временем, отображения схем могут стать некорректными в результате изменений структуры или ограничений. Поэтому важно уметь находить некорректные отображения схем и адаптировать их к новой структуре или ограничениям.

Вообще говоря, желательно автоматически находить некорректные отображения схем, поскольку сложность схем и количество отображений схем, используемых в приложениях, увеличиваются. Автоматическая или полуавтоматическая адаптация отображений к изменениям схем также желательна. Следует отметить, что автоматическая адаптация отображений схем – не то же самое, что автоматическое сопоставление схем. Задача адаптации схем – разрешить семантические соответствия, используя известные изменения внутрисхемной семантики, семантику существующих отображений и обна-

руженные семантические несогласованности (образовавшиеся в результате изменения схем). Сопоставление схем по необходимости ближе к генерированию отображений схем «с нуля», в этом случае мы лишены возможности (или роскоши) пользоваться такими знаниями о контексте.

Обнаружение некорректных отображений

В общем случае некорректные отображения, появившиеся в результате изменения схем, можно обнаруживать заблаговременно или постфактум. В первом случае отображения схем проверяются на согласованность сразу после того, как пользователь внес в схему изменения. Предполагается (или требуется), что система обслуживания отображений узнает обо всех изменениях схемы сразу после их совершения. Например, в системе ToMAS предполагается, что пользователи вносят в схемы изменения, пользуясь встроенными в систему редакторами, так что система сразу же узнает об изменениях. После того как изменения обнаружены, некорректные отображения можно найти, выполнив семантическую трансляцию существующих отображений с помощью логических отношений обновленной схемы.

Во втором случае система обслуживания отображений не знает о том, какие схемы были изменены и когда. Для обнаружения некорректностей все отображения схем регулярно проверяются, для чего выполняются запросы к источникам данных, и результаты транслируются с использованием существующих отображений. По результатам этих проверок выявляются некорректные отображения.

Альтернативный метод обнаружения некорректных отображений – использовать машинное обучение (как в системе Maveric). Было предложено построить ансамбль обученных *сенсоров* (по аналогии с несколькими обучаемыми при сопоставлении схем), распознающих некорректные отображения. Примерами могут служить сенсоры для мониторинга характеристик распределения значений целевых экземпляров, сенсоры для мониторинга средней частоты модификации данных и сенсоры структуры и ограничений, которые сравнивают транслированные данные с ожидаемыми синтаксисом и семантикой целевой схемы. Затем вычисляется взвешенная сумма показаний отдельных сенсоров, причем веса тоже подлежат обучению. Если результат указывает на присутствие изменений и последующие проверки подтверждают, что это действительно возможно, то генерируется оповещение.

Адаптация некорректных отображений

Выявленные некорректные отображения схем необходимо адаптировать к изменениям схем, восстановив корректность. Были предложены различные подходы к высокоуровневой адаптации схем. Среди них можно выделить три широких класса. В *подходах с фиксированным правилом* для каждого типа ожидаемых изменений схем определяется правило изменения отображения. В *подходах с мостиковым отображением* исходная схема S сравнивается с измененной схемой S' и генерируется новое отображение из S в S' в дополнение к существующим. В *подходах с семантической перезаписью* на основе семантической информации, закодированной в существующих отображениях, схемах

и внесенных в схемы семантических изменениях, предлагается переписать отображения, так чтобы получались семантически согласованные целевые данные. В большинстве случаев возможно несколько способов такой перезаписи, поэтому необходимо ранжировать кандидатов, прежде чем предъявить их пользователю для принятия окончательного решения (основанного на соображениях, зависящих от сценария или требований бизнеса, которые не отражены в схемах или отображениях).

Полную перестройку отображения схем (с чистого листа, начиная с сопоставления схем) тоже можно назвать вариантом адаптации. Но в большинстве случаев перезапись отображений обходится дешевле повторного генерирования, потому что при перезаписи можно воспользоваться знаниями, закодированными в существующих отображениях, и не вычислять отображения, которые в любом случае будут отвергнуты пользователем.

7.1.5. Очистка данных

Ошибки в исходных базах данных обязательно случаются, поэтому важно произвести очистку, чтобы правильно отвечать на запросы пользователей. Очистка данных – проблема, которая возникает и в хранилищах данных, и в системах интеграции, но контексты различаются. В хранилищах данные фактически извлекаются из локальных операционных баз данных и материализуются в виде глобальной базы данных, поэтому очистка производится в процессе создания глобальной базы. В системах интеграции данных очистка – это процесс, который следует производить на этапе обработки запроса, когда данные возвращаются из исходных баз.

Ошибки, которые призвана устранить очистка данных, можно отнести к двум категориям: уровня схемы и уровня экземпляров. Ошибки уровня схемы возникают в каждой конкретной ЛКС из-за нарушения явных или неявных ограничений. Например, значения атрибутов могут выходить за пределы, диктуемые областью их определения (скажем, 14-й месяц или отрицательная зарплата), они могут нарушать неявные зависимости (возраст может не соответствовать разности между текущей датой и датой рождения), возможно нарушение ограничений уникальности или ссылочной целостности. Кроме того, в контексте, рассматриваемом в этой главе, гетерогенность ЛКС на уровне схем (структурная и семантическая) тоже может рассматриваться как проблема, нуждающаяся в разрешении. Ясно, что ошибки уровня схем необходимо выявлять на этапе сопоставления схем и исправлять во время интеграции схем.

Ошибки уровня экземпляров встречаются в самих данных. Например, у некоторых атрибутов могут отсутствовать значения, хотя они обязательны. Возможно неправильное написание и неправильный порядок слов (например, «M.D. Mary Smith» вместо «Mary Smith, M.D.»), различия в аббревиатурах (например, в одной исходной базе «J. Doe», а в другой – «J. N. Doe»), вложенные значения (например, составной адрес, включающий все вместе: улицу и номер дома, город, область и почтовый код), значения, по ошибке помещенные в другие поля, повторяющиеся значения, противоречивые значения

(в одной базе данных хранится одна величина зарплаты, а в другой – другая). Для исправления ошибок уровня экземпляров требуется генерировать отображения, так чтобы данные очищались во время выполнения функций отображения (запросов).

Популярный подход к очистке данных – определить ряд операторов, работающих со схемами или с данными. Операторы могут быть включены в план очистки данных. Примеры операторов уровня схемы – добавить или удалить столбец таблицы, изменить структуру таблицы, объединив некоторые столбцы или разбив столбец на два, или создать более сложные преобразования схем с помощью обобщенного оператора «отобразить», который принимает одно отношение и порождает одно или несколько отношений. Приведем несколько примеров операторов уровня данных: применение функции к каждому значению одного атрибута, объединение значений двух атрибутов в значение одного атрибута и обратный ему оператор расщепления, оператор сопоставления, который вычисляет приближенное соединение кортежей двух отношений, оператор кластеризации, который разбивает кортежи на группы и сворачивает кортежи, принадлежащие каждой группе, в один, применяя какой-то вид агрегирования, а также базовые операторы поиска и устранения дубликатов. Многие операторы уровня данных сравнивают отдельные кортежи двух отношений (из одной или разных схем) и решают, представляют ли они один и тот же факт. Это похоже на то, что делается при сопоставлении схем, только применяется к отдельным элементам данных, и при этом рассматриваются не значения индивидуальных атрибутов, а целые кортежи. Но и в этом контексте можно использовать те же методы, что описывались в разделе о сопоставлении схем (например, редакционное расстояние или значение soundex). Предлагались также специальные методы для эффективной работы в контексте очистки данных, например нечеткое сопоставление, когда вычисляется функция сходства, определяющая, являются ли два кортежа идентичными или хотя бы приблизительно похожими.

Принимая во внимание большой объем обрабатываемых данных, очистка на уровне данных обходится дорого, поэтому эффективность очень важна. К физической реализации каждого из вышеупомянутых операторов следует относиться очень придирчиво. В случае хранилищ данных очистку можно выполнять автономно в пакетном процессе, но в системах интеграции данных это иногда приходится делать оперативно в процессе выборки данных из источников. Конечно, в таком случае производительность очистки данных более критична.

7.2. Обработка мультибазовых запросов

Теперь обратимся к вопросу о доступе к интегрированной базе данных, полученной применением описанных в предыдущем разделе методов, – так называемой проблеме мультибазовых запросов. Как уже отмечалось, многие методы распределенной обработки и оптимизации запросов, обсуждавшиеся

в главе 4, переносятся на мультибазовые системы, но есть и существенные отличия. Напомним, что мы выделили четыре шага распределенной обработки запроса: декомпозиция запроса, локализация данных, глобальная оптимизация и локальная оптимизация. Сама природа мультибазовых систем диктует несколько иные шаги и другие методы. Составляющие СУБД могут быть автономными и поддерживать различные языки базы данных и средства обработки запросов. Таким образом, для эффективного взаимодействия с составляющими СУБД необходим уровень СУМБД (рис. 1.12), а это требует дополнительных шагов обработки запроса (рис. 7.13). Кроме того, составляющих СУБД может быть много, и каждая может обладать своим поведением, что предъявляет новые требования к разработке более адаптивных методов обработки запросов.

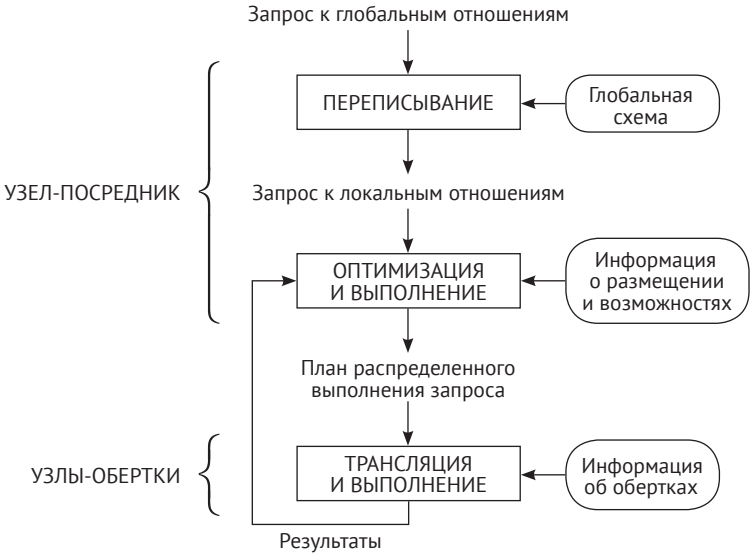


Рис. 7.13 ❖ Общая схема уровней для обработки мультибазовых запросов

7.2.1. Проблемы обработки мультибазовых запросов

Обработка запросов в мультибазовой системе сложнее, чем в распределенной СУБД, по следующим причинам:

1) вычислительные средства составляющих СУБД могут различаться, что мешает единообразной обработке запросов. Например, одни СУБД могут поддерживать сложные SQL-запросы с соединениями и агрегированием, а другие – нет. Таким образом, процессор мультибазовых запросов должен принимать во внимание разные возможности СУБД. Возможности каждой составляющей СУБД хранятся в каталоге вместе с информацией о размещении;

2) аналогично стоимость обработки запросов может быть различной в разных СУБД, как и возможности локальной оптимизации. Это увеличивает сложность подлежащих вычислению функций стоимости;
3) модели данных и языки, поддерживаемые составляющими СУБД, могут сильно различаться, например: реляционные, объектно-ориентированные, слабо структурированные и т. д. Это создает трудности при трансляции мультибазовых запросов на язык составляющей СУБД и объединении разнородных результатов;
4) поскольку мультибазовая система предоставляет доступ к очень разным СУБД с различной производительностью и поведением, методы распределенной обработки запросов должны адаптироваться к таким различиям.

Автономность составляющих СУБД ставит новые проблемы. Автономность СУБД можно определить по трем основным направлениям: взаимодействие, проектные решения и выполнение. Автономность по взаимодействию означает, что составляющая СУБД взаимодействует с другими по своему усмотрению и, в частности, может в любой момент завершить свои службы. Это требует методов обработки запросов, устойчивых к недоступности системы. Вопрос в том, как система отвечает на вопросы, когда ее часть недоступна с самого начала или становится недоступной в процессе выполнения. Автономность в силу проектного решения может ограничивать доступность и точность информации о стоимости, необходимой для оптимизации запроса. Трудность определения локальных функций стоимости – важный вопрос. Автономность выполнения мультибазовых систем затрудняет применение некоторых стратегий оптимизации запросов, рассмотренных в предыдущих главах. Например, оптимизация распределенных соединений, основанная на полусоединениях, может оказаться затруднительной, если исходное и целевое отношения находятся в разных составляющих СУБД, поскольку в этом случае выполнение соединения транслируется в три запроса: первый – выбрать значения атрибута соединения целевого отношения и отправить их в СУБД исходного отношения, второй – выполнить соединение на стороне исходного отношения и третий – выполнить соединение на стороне СУБД целевого отношения. Проблема возникает из-за того, что взаимодействие с составляющей СУБД имеет место на верхнем уровне API СУБД.

Помимо этих трудностей, некоторые проблемы ставит архитектура распределенной системы управления мультибазами данных. Архитектура, изображенная на рис. 1.12, показывает, в чем состоит дополнительная сложность. В распределенных СУБД процессоры запросов должны иметь дело только с распределением данных между несколькими узлами. С другой стороны, в распределенной СУМБД данные распределены не только между узлами, но и между несколькими базами данных, каждая из которых управляется автономной СУБД. Таким образом, количество сторон, участвующих в обработке запросов, возрастает с двух в распределенной СУБД (управляющий узел и локальные узлы) до трех в распределенной СУМБД: уровень СУМБД в управляющем узле (посредник) получает глобальный запрос, уровни СУМБД в узлах (обертки) участвуют в обработке запроса, а составляющие СУБД оптимизируют и исполняют запрос.

7.2.2. Архитектура обработки мультибазового запроса

Большая часть обработки мультибазового запроса производится в контексте архитектуры посредник–обертка (рис. 1.13). В этой архитектуре с каждой составляющей базой данных ассоциирована обертка, которая экспортирует информацию об исходной схеме, данных и возможностях обработки запросов. Посредник централизует информацию, полученную от оберток, создавая единое представление всех имеющихся данных (оно хранится в глобальном словаре), и выполняет обработку запроса, пользуясь обертками для доступа к составляющим СУБД. Модель данных посредника может быть реляционной, объектно-ориентированной или даже слабо структурированной. В этой главе мы сохраняем преемственность с предыдущими главами, посвященными распределенной обработке запросов, и продолжаем пользоваться реляционной моделью, которой вполне достаточно для объяснения методов обработки мультибазовых запросов.

У архитектуры посредник–обертка несколько достоинств. Во-первых, ее специализированные компоненты позволяют по-разному удовлетворять потребности различных видов пользователей. Во-вторых, посредники обычно специализируются на наборе взаимосвязанных составляющих баз данных, содержащих «похожие» данные, поэтому экспортируют схемы и семантику, характерные для конкретной предметной области. Специализация компонентов ведет к гибкой и расширяемой распределенной системе. В частности, она открывает возможность органичной интеграции различных данных, хранящихся в очень разных компонентах – от полноценной реляционной СУБД до простых файлов.

Предполагая использование архитектуры посредник–обертка, мы теперь можем обсудить различные уровни, участвующие в обработке запросов в распределенной мультибазовой системе, показанные на рис. 7.13. Как и раньше, будем считать, что на вход подается запрос к глобальным отношениям, выраженный средствами реляционного исчисления. Поскольку запрос адресован глобальным (распределенным) отношениям, распределенность и гетерогенность данных скрыты. В обработке мультибазового запроса принимают участие три основных уровня. Это разбиение на уровни похоже на то, что мы видели в однородных распределенных СУБД (рис. 4.2). Однако, поскольку никакой фрагментации нет, отпадает необходимость в уровне локализации данных.

Первые два уровня отображают входной запрос на оптимизированный план его распределенного выполнения (ПВЗ). Они осуществляют функции переписывания запроса, оптимизации запроса и частично выполнения запроса. Эти два уровня реализуются посредником, для чего используется метаинформация, хранящаяся в глобальном каталоге (о глобальной схеме, о размещении и о возможностях). Процедура переписывания запроса пользуется глобальной схемой и преобразует входной запрос в запрос к локальным отношениям. Напомним, что существует два основных подхода к интеграции базы данных: «глобальная как представление» (ГКП) и «локальная

как представление» (ЛКП). Таким образом, глобальная схема предоставляет определения представлений (т. е. отображения между глобальными отношениями и локальными отношениями, хранящимися в составляющих базах данных), и запросы переписываются с использованием этих представлений.

Переписывание можно производить на уровне реляционного исчисления или реляционной алгебры. В этой главе мы будем использовать обобщенную форму реляционного исчисления Datalog, хорошо приспособленную для такого переписывания. Таким образом, имеется дополнительный шаг трансляции исчисления в алгебру, напоминающий шаг декомпозиции в однородных распределенных СУБД.

Второй уровень выполняет оптимизацию запроса и (частично) его выполнение с учетом размещения локальных отношений и различных возможностей обработки запроса, наличествующих у составляющих СУБД и экспортируемых обертками. Информация о размещении и возможностях, используемая на этом уровне, может также содержать гетерогенные сведения о стоимости. Распределенный ПВЗ, порожденный этим уровнем, группирует внутри подзапросов операции, которые могут быть выполнены составляющими СУБД и обертками. Как и в распределенных СУБД, оптимизация запроса может быть статической или динамической. Однако из-за гетерогенности мультибазовых систем (например, некоторые составляющие СУБД могут возвращать ответы с неожиданно большой задержкой) динамическая оптимизация запросов может стать более критичной. В случае динамической оптимизации к этому уровню могут быть дополнительные обращения после выполнения следующего уровня, как показывают стрелки, обозначающие результаты, которые поступают от уровня трансляции и выполнения. Наконец, этот уровень объединяет результаты, поступающие от различных оберток, с целью дать унифицированный ответ на запрос пользователя. Для этого нужна возможность выполнять некоторые операции над данными, приходящими от оберток. Поскольку обертки могут обладать крайне ограниченными возможностями выполнения, например в случае очень простых составляющих СУБД, посредник должен самостоятельно предоставлять все возможности, необходимые для поддержки объявленного интерфейса.

Третий уровень осуществляет *трансляцию и выполнение запроса*, пользуясь обертками. Затем он возвращает результаты посреднику, который может объединить результаты от различных оберток. Каждая обертка поддерживает *схему обертки*, которая включает локальную экспортируемую схему и информацию об отображении, необходимую для трансляции входного подзапроса (подмножества ПВЗ) с общего языка на язык составляющей СУБД. Оттранслированный подзапрос выполняется составляющей СУБД, а локальный результат транслируется обратно в общий формат.

Информация, хранимая оберткой, описывает, как осуществить отображение из локальной схемы в глобальную и наоборот. Между базами данных возможны различные преобразования. Например, если в глобальной схеме температуры измеряются по шкале Фаренгейта, а в составляющих СУБД – по шкале Цельсия, то обертка может хранить формулу преобразования, благодаря которой глобальные пользователи и локальные базы данных видят информацию в ожидаемом виде. Если производится преобразование типов

и простых формул для этого недостаточно, то в каталоге обертки могут хранить полные перекодировочные таблицы.

7.2.3. Переписывание запросов с помощью представлений

Переписывание запроса – это процесс преобразования входного запроса к глобальным отношениям в запрос к локальным отношениям. Для этого используется глобальная схема, которая описывает в терминах представлений соответствия между глобальными и локальными отношениями. Таким образом, запрос необходимо переписать с помощью представлений. Методы переписывания запросов сильно различаются в зависимости от принятого подхода к интеграции баз данных: ЛКП или ГКП. В частности, методы, применяемые для ЛКП (и его обобщения ГЛКП), гораздо сложнее. Большая часть работы по переписыванию запросов с помощью представлений делается на логическом языке баз данных Datalog. Этот язык лаконичнее реляционного исчисления и потому более удобен для описания сложных алгоритмов переписывания запросов. В этом разделе мы познакомимся с терминологией Datalog. А затем опишем основные методы и алгоритмы переписывания запросов в подходах ЛКП и ГКП.

7.2.3.1. Терминология Datalog

Datalog можно рассматривать как встраиваемую версию реляционного исчисления доменов. Сначала определим *конъюнктивные запросы*, т. е. запросы вида выборка–проекция–соединение, которые являются основой для более сложных запросов. В Datalog конъюнктивный запрос можно выразить в виде правила вида

$$Q(t) : -\mathtt{R}_1(t_1), \ldots, \mathtt{R}_n(t_n).$$

Атом $Q(t)$ является *головой* запроса и обозначает результирующее отношение. Атомы $\mathtt{R}_1(t_1), \ldots, \mathtt{R}_n(t_n)$ – это *подцели* запроса, они обозначают отношения в базе данных. Q и $\mathtt{R}_1, \ldots, \mathtt{R}_n$ – имена предикатов, соответствующие именам отношений. t, $t_1, \ldots, t_n$ – кортежи отношений, они содержат переменные или константы. Переменные – аналоги доменных переменных в реляционном исчислении доменов. Таким образом, использование одного и того же имени переменной в нескольких предикатах выражает предикаты эквисоединения. Константы соответствуют предикатам равенства. Более сложные предикаты сравнения (например, содержащие операторы $\neq, \leqslant, <$) должны выражаться как другие подцели. Мы рассматриваем *безопасные* запросы, т. е. такие, что каждая переменная, встречающаяся в голове, встречается также и в теле. Дизъюнктивные запросы можно выразить в Datalog с помощью объединений, т. е. нескольких конъюнктивных запросов с одним и тем же головным предикатом.

Пример 7.12. Рассмотрим отношения `EMP` и `WORKS` из ГКС, определенные на рис. 7.9, и следующий SQL-запрос:

```
SELECT E#, TITLE, P#
FROM   EMP NATURAL JOIN WORKS
WHERE  TITLE = "Programmer" OR DUR = 24
```

На Datalog соответствующий запрос можно выразить в виде:

Q(E#, TITLE, P#) : —EMP(E#, ENAME, "Programmer", CITY),
WORKS(E#, P#, RESP, DUR)

Q(E#, TITLE, P#) : —EMP(E#, ENAME, TITLE, CITY),
WORKS(E#, P#, RESP, 24) ♦

7.2.3.2. Переписывание в случае ГКП

В подходе ГКП глобальная схема выражается в терминах источников данных, и каждое глобальное отношение определяется как представление над локальными отношениями. Это похоже на определение глобальной схемы в тесно интегрированных распределенных СУБД. В частности, локальные отношения (т. е. отношения в составляющих СУБД) можно уподобить фрагментам. Однако поскольку локальные базы данных уже существуют и автономны, может случиться, что кортеж глобального отношения не встречается в локальных или встречается в разных локальных отношениях. Поэтому свойства полноты и дизъюнктности фрагментации гарантировать невозможно. Из-за отсутствия полноты ответы на запросы могут быть неполны. А отсутствие дизъюнктности может стать причиной дубликатов в результате, которые тем не менее несут полезную информацию и не должны удаляться. Как и в запросах, в определениях представлений может использоваться нотация Datalog.

Пример 7.13. Рассмотрим глобальные отношения EMP и WORKS на рис. 7.9 с небольшой модификацией: по умолчанию обязанности работника в проекте определяются его должностью, поэтому атрибут TITLE присутствует в отношении WORKS, но отсутствует в отношении EMP. Рассмотрим локальные отношения EMP1 и EMP2 с атрибутами E#, ENAME, TITLE и CITY каждое и локальное отношение WORKS1 с атрибутами E#, P# и DUR. Глобальные отношения EMP и WORKS можно определить с помощью следующих правил Datalog:

EMP(E#, ENAME, CITY): —EMP1(E#, ENAME, TITLE, CITY) (d_1)
EMP(E#, ENAME, TITLE, CITY): —EMP2(E#, ENAME, TITLE, CITY) (d_2)
WORKS(E#, P#, TITLE, DUR): —EMP1(E#, ENAME, TITLE, CITY),
—WORKS1(E#, P#, DUR) (d_3)
WORKS(E#, P#, TITLE, DUR): —EMP2(E#, ENAME, TITLE, CITY)),
—WORKS1(E#, P#, DUR) (d_4)
♦

Переписать запрос, выраженный относительно глобальной схемы, в эквивалентный запрос к локальным отношениям сравнительно просто, это напоминает локализацию данных в тесно интегрированных распределенных СУБД (см. раздел 4.2). Техника переписывания с использованием представлений называется *разворачиванием* (unfolding) – каждое глобальное отношение, упоминаемое в запросе, заменяется соответствующим ему представлением.

Для этого правила определения представлений применяются к запросу, в результате чего получается объединение конъюнктивных запросов, по одному для каждого применения правила. Поскольку глобальное отношение может быть определено несколькими правилами (см. пример 7.13), разворачивание может порождать избыточные запросы, которые следует исключить.

Пример 7.14. Рассмотрим глобальную схему из примера 7.13 и следующий запрос q, в котором мы хотим получить информацию о назначении на проекты для работников, проживающих в Париже:

$Q(e, p)$: –EMP(e, ENAME, "Paris"), WORKS(e, p, TITLE, DUR).

Разворачивание q порождает q' следующим образом:

$Q'(e, p)$: –EMP1(e, ENAME, TITLE, "Paris"), WORKS1(e, p, DUR). (q_1)
$Q'(e, p)$: –EMP2(e, ENAME, TITLE, "Paris"), WORKS1(e, p, DUR). (q_2)

Q' является объединением двух конъюнктивных запросов: q_1 и q_2. Запрос q_1 получается применением правила ГКП d_3 или обоих правил $d1$ и d_3. Во втором случае полученный запрос избыточен относительно того, который получен применением только d_3. Аналогично запрос q_2 получается применением правила d_4 или обоих правил d_2 и d_4. ♦

Хотя базовая техника проста, переписывание в ГКП осложняется, когда паттерны доступа к локальным базам данных ограничены. Так обстоит дело для баз данных, доступных через веб, когда к отношениям можно обратиться только с помощью определенных паттернов привязки для их атрибутов. В таком случае простой подстановки представлений вместо глобальных отношений недостаточно, и для переписывания запроса приходится прибегать к рекурсивным запросам Datalog.

7.2.3.3. Переписывание в случае ЛКП

В подходе ЛКП глобальная схема выражается независимо от локальных баз данных, и каждое локальное отношение определяется как представление над глобальными отношениями. Это обеспечивает большую гибкость в определении локальных отношений.

Пример 7.15. Чтобы было проще сравнивать с ГКП, приведем пример, симметричный примеру 7.13. Теперь EMP и WORKS – исходные глобальные отношения, а локальные отношения EMP1, EMP2 и WORKS1 определены следующими правилами Datalog:

```
EMP1(E#, ENAME, TITLE, CITY) : –EMP(E#, ENAME, CITY),
                                WORKS(E#, P#, TITLE, DUR)          (d5)
EMP2(E#, ENAME, TITLE, CITY) : –EMP(E#, ENAME, CITY),
                                WORKS(E#, P#, TITLE, DUR)          (d6)
         WORKS1(E#, P#, DUR): –WORKS(E#, P#, TITLE, DUR)           (d7)
```

♦

Переписать запрос, выраженный относительно глобальной схемы, в эквивалентный запрос к представлениям, описывающим локальные отношения, трудно по трем причинам. Во-первых, в отличие от ГКП, нет прямого соответствия между термами, используемыми в глобальной схеме (например, EMP, ENAME), и теми, что используются в представлениях (например, EMP1, EMP2, ENAME). Для отыскания соответствий необходимо сравнение с каждым представлением. Во-вторых, представлений может быть гораздо больше, чем глобальных отношений, поэтому сравнение с ними занимает много времени. В-третьих, определения представлений могут содержать сложные предикаты, отражающие специфику локальных отношений, например представление EMP3 содержит только программистов. Таким образом, не всегда возможно переписать запрос в эквивалентной форме. В таком случае лучшее, на что можно рассчитывать, – найти *максимальный включенный* запрос, т. е. запрос, порождающий максимальное подмножество истинного ответа. Например, EMP3 могло бы вернуть только подмножество всех работников, являющихся программистами.

Переписывание запросов с помощью представлений вызвало значительный интерес из-за его связей с проблемами логической и физической интеграции данных. В контексте физической интеграции (т. е. хранилищ данных) использование материализованных представлений может оказаться гораздо эффективнее доступа к базовым отношениям. Однако задача нахождения переписывания с использованием представлений NP-полная относительно количества представлений и количества подцелей запроса. Поэтому алгоритмы стремятся уменьшить число рассматриваемых вариантов переписывания. Для решения задачи предложено три основных алгоритма: алгоритм корзин, алгоритм обратного правила и алгоритм MinCon. У алгоритмов корзин и обратного правила похожие ограничения, которые снимаются алгоритмом MinCon.

В алгоритме корзин каждый предикат запроса рассматривается независимо, чтобы отобрать только релевантные этому предикату представления. При заданном запросе Q алгоритм состоит из двух шагов. На первом шаге строится корзина b для каждой подцели q запроса Q, которая не является предикатом сравнения, и в b вставляются головы представлений, релевантных ответу на q. Чтобы представление V находилось в b, должно существовать отображение, отождествляющее q с одной подцелью v в V.

Например, рассмотрим запрос Q в примере 7.14 и представления в примере 7.15. Следующее отображение отождествляет подцель EMP(e, ENAME, "Paris") запроса Q с подцелью EMP(E#, ENAME, CITY) в представлении EMP1:

$e \rightarrow$ E#, "Paris" $\rightarrow$ CITY.

На втором шаге для каждого представления V декартова произведения непустых корзин (т. е. некоторого подмножества корзин) алгоритм порождает конъюнктивный запрос и проверяет, содержится ли он в Q. Если да, то этот конъюнктивный запрос запоминается, поскольку он представляет один способ ответить на часть Q из V. Перезаписанный запрос является объединением конъюнктивных запросов.

Пример 7.16. Рассмотрим запрос Q из примера 7.14 и представления из примера 7.15. На первом шаге алгоритм корзин создает две корзины, по одной для каждой подцели Q. Обозначим b_1 корзину для подцели EMP(e, ENAME, "Paris") и b_2 корзину для подцели WORKS(e, p, TITLE, DUR). Поскольку алгоритм вставляет в корзину только головы представлений, то в голове представления могут быть переменные, которых нет в отождествляющем отображении. Такие переменные просто помечаются штрихами. Получаем следующие корзины:

b_1 = {EMP1(E#, ENAME, TITLE′, CITY),
EMP2(E#, ENAME, TITLE′, CITY),}

b_2 = {WORKS1(E#, P#, DUR)}

На втором шаге алгоритм комбинирует элементы из корзин, что дает объединение двух конъюнктивных запросов:

$Q'(e, p)$: −EMP1(e, ENAME, TITLE, "Paris"), WORKS1(e, p, DUR) (q_1)
$Q'(e, p)$: −EMP2(e, ENAME, TITLE, "Paris"), WORKS1(e, p, DUR) (q_2)

♦

Главное достоинство алгоритма корзин в том, что благодаря рассмотрению предикатов запроса он позволяет значительно уменьшить количество просматриваемых переписываний. Однако если рассматривать предикаты по отдельности, то в корзину может быть добавлено представление, которое окажется нерелевантным при рассмотрении соединения с другими представлениями. Кроме того, на втором шаге все равно может порождаться много переписываний, поскольку строится декартово произведение корзин.

Пример 7.17. Рассмотрим запрос Q из примера 7.14 и представления из примера 7.15 с добавлением следующего представления, которое возвращает проекты, для которых существуют работники, проживающие в Париже.

PROJ1(P#): −EMP1(E#, ENAME, "Paris"),
WORKS(E#, P#, TITLE, DUR) (d_8)

Тогда следующее отображение отождествляет подцель WORKS(e, p, TITLE, DUR) запроса Q с подцелью WORKS(E#, P#, TITLE, DUR) представления PROJ1:

$p \rightarrow$ PNAME.

Таким образом, на первом шаге алгоритм корзин добавляет PROJ1 в корзину b_2. Однако PROJ1 не может быть полезно для переписывания Q, т. к. переменная ENAME отсутствует в голове PROJ1, поэтому соединить PROJ1 по переменной e из Q невозможно. Но это можно обнаружить только на втором шаге, при построении конъюнктивных запросов. ♦

Алгоритм MinCon устраняет ограничения алгоритма корзин (и алгоритма обратного правила), поскольку запрос анализируется глобально и рассматривается, как каждый предикат запроса взаимодействует с представлениями. Как и алгоритм корзин, он состоит из двух шагов. На первом шаге отбираются представления, которые содержат подцели, соответствующие подцелям за-

проса *Q*. Но после того как отображение, отождествляющее подцель *q* запроса *Q* с подцелью *v* представления V, найдено, алгоритм рассматривает предикаты соединения в *Q* и находит минимальное множество дополнительных подцелей *Q*, которые необходимо отобразить на подцели V. Это множество подцелей *Q* запоминается в *MinCon-описании* (MinCon description – MCD), ассоциированном с V. На втором шаге порождается переписанный запрос путем комбинирования различных MCD. В отличие от алгоритма корзин, на втором шаге не нужно проверять, что предложенные переписывания содержатся в запросе, потому что сам способ создания MCD гарантирует, что результирующие переписывания будут содержаться в исходном запросе.

В примере 7.17 алгоритм создал бы 3 MCD: два для представлений EMP1 и EMP2, содержащие подцель EMP запроса *Q*, и один для представления ASG1, содержащий подцель ASG. Однако алгоритм не может создать MCD для PROJ1, потому что не может применить предикат соединения в *Q*. Таким образом, алгоритм сгенерировал бы переписанный запрос *Q* из примера 7.16. По сравнению с алгоритмом корзин, второй шаг алгоритма MinCon гораздо эффективнее, потому что просматривает меньше комбинаций MCD.

7.2.4. Оптимизация и выполнение запроса

Три основные проблемы оптимизации запросов в мультибазовых системах – моделирование гетерогенной стоимости, гетерогенная оптимизация запроса (с учетом разных возможностей составляющих СУБД) и адаптивная обработка запроса (с учетом значительных вариаций в поведении среды: отказов, непредсказуемых задержек и т. п.). В этом разделе мы опишем методы решения первых двух проблем. В разделе 4.6 были представлены методы адаптивной обработки запросов. Эти методы можно использовать также в мультибазовых системах, при условии что обертки умеют собирать информацию о выполнении внутри составляющих СУБД.

7.2.4.1. Моделирование гетерогенной стоимости

Определение глобальной функции стоимости и связанная с этим задача получения относящейся к стоимости информации из составляющих СУБД – пожалуй, наиболее хорошо изученная из трех поставленных выше проблем. Предложено несколько решений, которые мы обсудим ниже.

Прежде всего отметим, что нас в основном интересует определение стоимости нижних уровней дерева выполнения запроса, соответствующих тем его частям, которые выполняются составляющими СУБД. Если предположить, что вся локальная обработка «спущена» вниз дерева, то план выполнения запроса можно модифицировать, так чтобы листья дерева соответствовали подзапросам, которые будут выполняться составляющими СУБД. В этом случае речь идет об определении стоимости этих подзапросов, подаваемых на вход операторов первого уровня (считая снизу). Стоимость более высоких уровней дерева выполнения запроса можно вычислить рекурсивно, зная стоимость листовых узлов.

К определению стоимости выполнения запросов составляющими СУБД существует три подхода.

1. **Подход на основе черного ящика.** В этом случае каждая составляющая СУБД рассматривается как черный ящик, в котором выполняются некоторые тестовые запросы, и на основе их результатов определяется необходимая информация о стоимости.
2. **Специализированный подход.** В этом случае на основе уже имеющихся знаний о составляющих СУБД, а также их внешних характеристик субъективно определяется информация о стоимости.
3. **Динамический подход.** В этом случае отслеживается поведение составляющих СУБД во время выполнения, и информация о стоимости собирается динамически.

Мы обсудим все три подхода, останавливаясь на предложениях, вызвавших наибольший интерес.

Подход на основе черного ящика

В этом случае функции стоимости выражаются логически (например, агрегированные стоимости процессора и ввода-вывода, коэффициенты избирательности), а не на основе физических характеристик (например, мощности отношений, количества страниц, количества уникальных значений в каждом столбце). Таким образом, функции стоимости для составляющих СУБД имеют вид:

Стоимость = стоимость инициализации + стоимость нахождения подходящих кортежей + стоимость обработки выбранных кортежей.

Слагаемые в этой формуле зависят от операторов. Однако эти различия нетрудно задать априори. Настоящая трудность – определить коэффициенты при членах этой формулы, которые различны в разных составляющих СУБД. Один из способов решить эту проблему – построить искусственную базу данных (называемую *калибровочной*), выполнить в ней изолированные запросы и, измерив время, вывести коэффициенты.

Беда в том, что результаты, полученные на искусственной базе данных, могут не иметь никакого отношения к реальной. Альтернатива – выполнять пробные запросы в составляющих СУБД и так добывать информацию о стоимости. Пробные запросы можно использовать для сбора сведений о многих факторах, влияющих на стоимость. Например, с их помощью можно извлечь из составляющих СУБД данные, необходимые для построения и обновления мультибазового каталога. Статистические пробные запросы позволяют узнать количество кортежей в отношении. Наконец, пробные запросы с замером времени дают возможность оценить производительность и ее вклад в коэффициенты функции стоимости.

Частным случаем пробных запросов являются выборочные запросы. В этом случае запросы классифицируются по различным критериям, и выполнение выборочных запросов из каждого класса хронометрируется для получения частичной информации о стоимости. Классификация может быть основана

на характеристиках запроса (например, унарные запросы и запросы с соединением двух отношений), на характеристиках отношений-операндов (например, мощность, количество атрибутов, сведения об индексированных атрибутах) и на характеристиках составляющих СУБД (например, поддерживаемые методы доступа и стратегии выбора методов доступа).

Определяются правила классификации для идентификации запросов, которые выполняются похожим образом и, следовательно, имеют общую формулу стоимости. Например, можно рассмотреть два запроса с похожими алгебраическими выражениями (т. е. одинаковой формой алгебраического дерева), но разными отношениями-операндами, атрибутами и константами, и предположить, что они выполняются одинаково, если физические свойства атрибутов одинаковы. Другой пример – можно предположить, что порядок соединения, указанный в запросе, не влияет на выполнение, поскольку оптимизатор выбирает самый эффективный порядок соединения, самостоятельно осуществляя переупорядочение. Таким образом, два запроса, в которых соединяется один и тот же набор отношений, принадлежат одному классу, какой бы порядок ни указал пользователь. Правила классификации комбинируются для определения классов запросов. Классификация производится сверху вниз – тогда каждый класс разбивается на более узкие – или снизу вверх – тогда два класса объединяются в один. На практике наиболее эффективная классификация получается в результате совместного применения обоих подходов. Глобальная функция стоимости состоит из трех компонентов: стоимость инициализации, стоимость извлечения кортежа и стоимость обработки кортежа. Разница в том, как определяются параметры этой функции. Вместо использования калибровочной базы данных выполняются выборочные запросы и измеряется стоимость. Уравнение глобальной стоимости рассматривается как уравнение регрессии, а коэффициенты регрессии вычисляются на основе измеренных стоимостей выполнения выборочных запросов. Коэффициенты регрессии и есть параметры функции стоимости. В конечном итоге качество модели стоимости контролируется с помощью статистических критериев (например, критерия Фишера): если критерий не выполняется, то классификация запросов уточняется, и так до тех пор, пока качество не будет признано достаточно хорошим.

В описанных выше подходах необходим предварительный шаг – инициализация модели стоимости (путем калибровки или выборки). Это не всегда приемлемо, потому что привело бы к замедлению системы при добавлении каждой новой составляющей СУБД. Одно из возможных решений – постепенно обучать модель стоимости на запросах. Предполагается, что посредник обращается к составляющей СУБД посредством вызова функции. Стоимость вызова складывается из трех величин: время до получения первого кортежа, общее время получения результата и мощность результата. Это позволяет оптимизатору запросов минимизировать либо время до получения первого кортежа, либо время обработки всего запроса – в зависимости от требований пользователя. Первоначально процессор запросов не имеет никакой статистики составляющих СУБД. Затем он начинает наблюдать за проходящими запросами: получает время обработки каждого вызова и запоминает его для оценки в будущем. Для управления большим объемом собранной статистики

менеджер стоимости обобщает ее – либо вообще без потери точности, либо жертвуя точностью ради уменьшения занятой памяти и ускорения оценки стоимости. Обобщение сводится к агрегированию статистики: вычисляется среднее время всех вызовов одного вида, т. е. с одинаковым именем функции и нулем или более одинаковыми значениями аргументов. Модуль оценки стоимости реализован на декларативном языке. Это позволяет добавлять новые формулы стоимости, описывающие поведение конкретных составляющих СУБД. Однако ответственность за расширение модели стоимости посредника по-прежнему возлагается на разработчика посредника.

Основной недостаток подхода на основе черного ящика заключается в том, что модель стоимости, пусть и корректируемая в результате калибровки, одинакова для всех составляющих СУБД и может не улавливать специфики каждой из них. Поэтому оценка стоимости выполнения запроса каждой СУБД может оказаться неточной, что приведет к неожиданному поведению.

Специализированный подход

Предположение, лежащее в основе этого подхода, заключается в том, что процессоры запросов в составляющих СУБД слишком сильно различаются и потому не могут быть представлены единой моделью стоимости, как в подходе на основе черного ящика. Также предполагается, что умение точно оценить стоимость локальных подзапросов положительно скажется на глобальной оптимизации запроса. Подход предлагает инфраструктуру для интеграции моделей стоимости составляющих СУБД в оптимизатор запросов на стороне посредника. Решение состоит в том, чтобы расширить интерфейс оберток, дав посреднику возможность получать от них специфическую информацию о стоимости. Разработчик обертки вправе предоставить модель стоимости, полностью или частично. Таким образом, проблема в том, чтобы интегрировать эти (возможно, частичные) описания стоимости в оптимизатор запросов на стороне посредника. Существует два решения.

Первое – включить в состав обертки логику вычисления трех оценок стоимости: время инициализации обработки запроса и получения первого результата (стоимость сброса *reset_cost*), время получения следующего результата (стоимость продвижения *advance_cost*) и мощность результата *cardinality*. А общая стоимость обработки запроса тогда равна

$$Total\_access\_cost = reset\_cost + (cardinality - 1) * advance\_cost.$$

Это решение можно обобщить для оценки стоимости вызова процедур базы данных. В таком случае обертка предоставляет формулу стоимости, линейно зависящую от параметров процедуры. Это решение было успешно реализовано для моделирования широкого круга гетерогенных составляющих СУБД – от реляционных до сервера образов. Показано, что нужно совсем немного усилий, чтобы реализовать простую модель стоимости и тем значительно улучшить обработку распределенных запросов к гетерогенным источникам.

Второе решение – воспользоваться иерархической обобщенной моделью стоимости. Как показано на рис. 7.14, каждый узел представляет правило,

ассоциирующее с паттерном запроса функцию стоимости для различных стоимостных параметров.

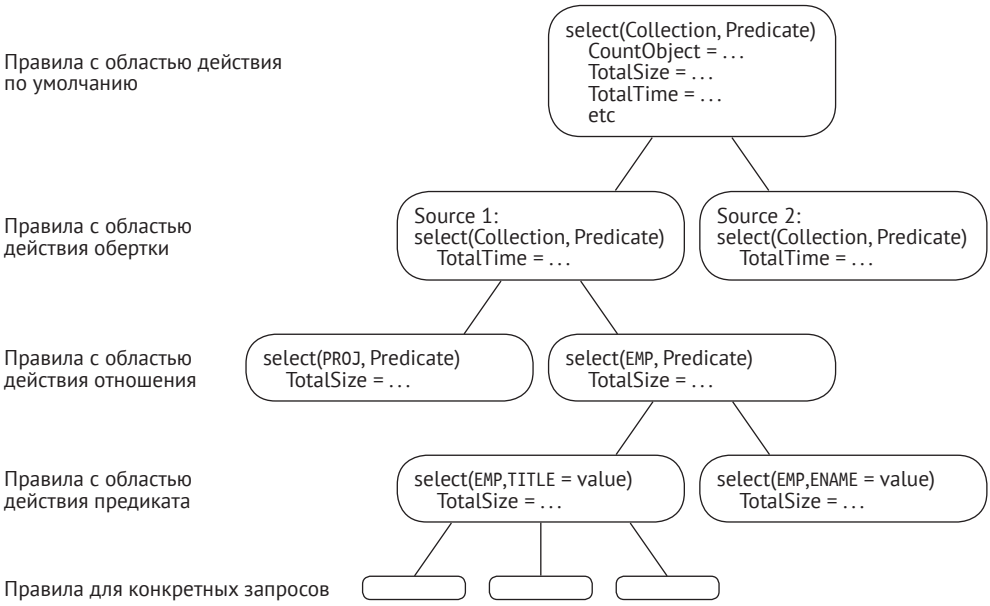


Рис. 7.14 ❖ Иерархическое дерево модели стоимости

Иерархия узлов насчитывает пять уровней, зависящих от общности правил (на рис. 7.14 ширина блока тем больше, чем шире область применимости правил). Правила на верхнем уровне по умолчанию применимы к любой СУБД. Правила на следующих уровнях становятся уже: конкретная СУБД, отношение, предикат, запрос. На этапе регистрации обертки посредник получает от нее метаданные, включающие информацию о стоимости, и дополняет свою встроенную модель стоимости, добавляя в нее новые узлы на соответствующем уровне иерархии. Эта система достаточно общая, чтобы уловить и интегрировать как общие знания о стоимости, выраженные разработчиками оберток в виде правил, так и конкретную информацию, выведенную на основе ранее выполненных и запомненных запросов. Таким образом, благодаря иерархии наследования стоимостной оптимизатор в посреднике может поддержать широкий спектр источников данных. Посредник пользуется специализированной информацией о стоимости для каждой составляющей СУБД, чтобы точнее оценивать стоимость запросов и выбирать более эффективный ПВЗ.

Пример 7.18. Рассмотрим отношения `EMP` и `WORKS` из ГКС (рис. 7.9). `EMP` хранится в составляющей СУБД db_1 и содержит 1000 кортежей. `WORKS` хранится в составляющей СУБД db_2 и содержит 10 000 кортежей. Предполагается, что значения атрибутов распределены равномерно. В половине кортежей `WORKS` атрибут `DUR` меньше 6. Ниже приведено детальное описание некоторых частей обоб-

щенной модели стоимости в посреднике, где R и S – отношения, A – атрибут соединения, а верхние индексы обозначают метод доступа.

$cost(R) = |R|$.

$cost(\sigma_{predicate}(R)) = cost(R)$ (доступ к R путем последовательного просмотра – по умолчанию).

$cost(R \bowtie_A^{ind} S) = cost(R) + |R| * cost(\sigma_{A=v}(S))$ (использование соединения по индексу (*ind*) над S.A).

$cost(R \bowtie_A^{nl} S) = cost(R) + |R| * cost(S)$ (использование соединения методом вложенных циклов (*nl*)).

Рассмотрим следующий глобальный запрос *Q*:

```
SELECT *
FROM   EMP NATURAL JOIN WORKS
WHERE  WORKS.DUR>6
```

Стоимостной оптимизатор запросов генерирует следующие планы для обработки *Q*:

$P_1 = \sigma_{DUR>6}(EMP \bowtie_{E\#}^{ind} WORKS)$;

$P_2 = EMP \bowtie_{E\#}^{nl} \sigma_{DUR>6}(WORKS)$;

$P_3 = \sigma_{DUR>6}(WORKS) \bowtie_{E\#}^{ind} EMP$;

$P_4 = \sigma_{DUR>6}(WORKS) \bowtie_{E\#}^{nl} EMP$.

Основываясь на обобщенной модели стоимости, вычисляем их стоимость:

$$\begin{aligned} cost(P_1) &= cost(\sigma_{DUR>6}(EMP \bowtie_{E\#}^{nl} WORKS) \\ &= cost(EMP \bowtie_{E\#}^{ind} WORKS) \\ &= cost(EMP) + |EMP| * cost(\sigma_{E\#=v}(WORKS)) \\ &= |EMP| + |EMP| * |WORKS| = 10\ 001\ 000; \end{aligned}$$

$$\begin{aligned} cost(P_2) &= cost(EMP) + |EMP| * cost(\sigma_{DUR>6}(WORKS)) \\ &= cost(EMP) + |EMP| * cost(WORKS) \\ &= |EMP\ | + |EMP| * |WORKS| = 10\ 001\ 000; \end{aligned}$$

$$\begin{aligned} cost(P_3) &= cost(P_4) = |WORKS| + |WORKS|/2 * |EMP| \\ &= 5\ 010\ 000. \end{aligned}$$

Таким образом, оптимизатор отбрасывает планы P_1 и P_2, но сохраняет план *P*3 или *P*4 для обработки *Q*. Предположим, что посредник импортирует информацию о стоимости, специфичную для составляющих СУБД. db_1 экспортирует стоимость доступа к кортежам EMP в виде:

$cost(\sigma_{A=v}(R)) = |\sigma_{A=v}(R)|$.

db_2 экспортирует стоимость выборки из WORKS кортежей с заданным значением E#:

$cost(\sigma_{E\#=v}(WORKS)) = |\sigma_{E\#=v}(WORKS)|$.

Посредник включает эти функции стоимости в свою иерархическую модель стоимости и теперь может более точно оценить стоимость ПВЗ:

$$cost(P_1) = |\texttt{EMP}| + |\texttt{EMP}| * |\sigma_{\texttt{E\#}=v}(\texttt{WORKS})|$$
$$= 1000 + 1000 * 10$$
$$= 11\,000;$$
$$cost(P_2) = |\texttt{EMP}| + |\texttt{EMP}| * |\sigma_{\texttt{DUR}>6}(\texttt{WORKS})|$$
$$= |\texttt{EMP}| + |\texttt{EMP}| * |\texttt{WORKS}|/2$$
$$= 5\,001\,000;$$
$$cost(P_3) = |\texttt{WORKS}| + |\texttt{WORKS}| / 2 * |\sigma_{\texttt{E\#}=v}(\texttt{EMP})|$$
$$= 10\,000 + 5000 * 1$$
$$= 15\,000;$$
$$cost(P_4) = |\texttt{WORKS}| + |\texttt{WORKS}| / 2 * |\texttt{EMP}|$$
$$= 10\,000 + 5000 * 1000$$
$$= 5\,010\,000.$$

Теперь наилучшим является ПВЗ P_1, который ранее был отброшен из-за отсутствия информации о стоимости от составляющих СУБД. Во многих ситуациях P_1 – действительно лучший план для обработки Q_1. ♦

Оба описанных решения хорошо приспособлены к архитектуре посредник–обертка и предлагают неплохой компромисс между накладными расходами на предоставление информации о стоимости, специфичной для разных составляющих СУБД, и ускоренной обработкой гетерогенного запроса.

Динамический подход

В описанных выше подходах предполагается, что среда выполнения не изменяется со временем. Но в большинстве случаев это не так. В зависимости от динамичности изменения можно выделить три класса факторов среды выполнения. В первый класс входят часто изменяющиеся факторы (от одного раза в секунду до одного раза в несколько минут): нагрузка на процессоры, пропускная способность ввода-вывода и доступная память. Во второй класс входят медленно изменяющиеся факторы (от одного раза в час до одного раза в несколько дней): конфигурация параметров СУБД, физическая организация данных на диске и схема базы данных. В третий класс входят почти неизменные факторы (изменяются раз в несколько месяцев или ежегодно): тип СУБД, местоположение базы данных и быстродействие процессоров. Нас будут интересовать решения, разработанные для первых двух классов.

Один из способов адаптации к динамическим средам, в которых сетевой трафик, условия доступа к системе хранения или доступная память изменяются со временем, – обобщить выборочный метод и рассматривать пользовательские запросы как новые примеры. Измеренное время выполнения запроса используется для корректировки параметров модели прямо во время выполнения, а скорректированная модель применяется к последующим запросам. Это позволяет избежать накладных расходов на периодическую обработку выборочных запросов, но все равно требует сложных вычислений для решения уравнения модели стоимости и не гарантирует, что точность

модели будет улучшаться со временем. Есть решение лучше – качественное, когда определяется уровень состязания в системе как влияние комбинации часто изменяющихся факторов на стоимость обработки запроса. Уровень состязания в системе может принимать несколько дискретных значений: высокий, средний, низкий или нулевой. Это дает возможность построить многокатегорийную модель стоимости, которая дает точные оценки, хотя динамические факторы изменяются. В начальный момент модель стоимости калибруется с помощью пробных запросов. Текущий уровень состязания пересчитывается динамически на основе наиболее важных параметров системы. При таком подходе предполагается, что запросы короткие, так что факторы среды не успевают сильно измениться во время выполнения запроса. К длительным запросам это решение неприменимо.

Для случая, когда изменение факторов среды предсказуемо (например, нагрузка на СУБД каждый день варьируется одинаково), стоимость обработки запроса вычисляется для последовательных диапазонов дат. Тогда полная стоимость равна сумме стоимостей для каждого диапазона. Кроме того, иногда имеется возможность определить паттерн доступной пропускной способности сети между процессором запросов СУМБД и составляющей СУБД. Это позволяет корректировать стоимость обработки запроса в зависимости от даты.

7.2.4.2. Гетерогенная оптимизация запроса

Помимо гетерогенной модели стоимости, при оптимизации мультибазовых запросов нужно принимать во внимание гетерогенные вычислительные возможности составляющих СУБД. Например, одна СУБД может поддерживать только простые операции выборки, а другая – сложные запросы, включающие соединение и агрегирование. Поэтому в зависимости от того, как обертки экспортируют такие возможности, обработка запроса на стороне посредника может быть более или менее сложной. Существует два основных подхода к этой проблеме, зависящих от интерфейса между посредником и оберткой: на основе запросов и на основе операторов.

1. **Подход на основе запросов.** В этом случае обертки поддерживают одинаковые возможности запросов, например подмножество SQL, которые транслируются в возможности составляющей СУБД. Этот подход обычно опирается на стандартный интерфейс СУБД, например ODBC, и его расширения для оберток или на механизм SQL-управления внешними данными (SQL Management of External Data – SQL/MED). Таким образом, составляющие СУБД, с точки зрения посредника, однородны, поэтому можно использовать методы обработки запросов, предназначенные для однородных распределенных СУБД. Однако если средства составляющих СУБД ограничены, то недостающие возможности необходимо реализовать в самих обертках. Например, если СУБД не поддерживает соединений, то обрабатывать запросы с соединениями должна обертка.
2. **Подход на основе операторов.** В этом случае обертки экспортируют возможности составляющих СУБД с помощью композиции реляци-

онных операторов. Это повышает гибкость при определении уровня функциональности между посредником и оберткой. В частности, посреднику можно сделать доступными различные средства составляющих СУБД. Создание оберток при этом упрощается, но ценой более сложной обработки запросов на стороне посредника. В частности, функциональность, не поддерживаемую составляющими СУБД (то же соединение), должен реализовать посредник.

Далее в этом разделе мы подробно рассмотрим подходы к оптимизации запросов.

Подход на основе запросов

Поскольку составляющие СУБД представляются посреднику однородными, мы можем воспользоваться стоимостным алгоритмом оптимизации распределенных запросов (см. главу 4) с гетерогенной моделью стоимости (см. раздел 7.2.4.1). Однако необходимы расширения для преобразования распределенного плана выполнения в подзапросы, выполняемые составляющими СУБД, и подзапросы, выполняемые посредником. В этом случае полезен гибридный двухшаговый метод оптимизации (см. раздел 4.5.3): на первом шаге централизованный стоимостной оптимизатор порождает статический план, а на втором шаге, на этапе подготовки, создается план выполнения, для чего производится выбор узлов и размещение подзапросов в этих узлах. Однако централизованные оптимизаторы ограничивают пространство поиска, исключая из рассмотрения кустистые деревья соединения. Почти во всех системах используются леволинейные порядки соединения. Рассмотрение только леволинейных деревьев соединений дает хорошие результаты в централизованных СУБД по двум причинам: устраняется необходимость в оценке статистики по крайней мере для одного операнда, но все равно можно задействовать индексы для одного из операндов. Однако в мультибазовых системах эти типы планов выполнения соединений необязательно предпочтительные, потому что не допускают никакого распараллеливания соединений. В предыдущих главах мы говорили, что это проблема даже в однородных распределенных СУБД, но в мультибазовых системах она еще серьезнее, потому что мы хотим передать как можно больший объем обработки составляющим СУБД.

Чтобы разрешить эту проблему, нужно каким-то образом генерировать кустистые деревья соединений и рассматривать их вместо леволинейных. Один из способов сделать это – сначала применить стоимостной оптимизатор запросов, чтобы сгенерировать леволинейное дерево соединений, а затем преобразовать его в кустистое. В таком случае леволинейный план выполнения может быть оптимален по критерию полного времени, а преобразование улучшает время ответа на запрос, почти не затрагивая полное время. Возможен также гибридный алгоритм, который одновременно выполняет просмотр леволинейного дерева соединений снизу вверх и сверху вниз, осуществляя его пошаговое преобразование, если это возможно. Алгоритм поддерживает два указателя на узлы дерева, называемые *верхними якорными узлами* (upper anchor nodes – UAN). В начальный момент один из них, называемый нижним UAN (UAN_B), указывает на деда корневого узла (соединение

с R_3 на рис. 4.9), а второй, называемый верхним UAN (UAN_T), – на корень (соединение с R_5). Для каждого UAN алгоритм выбирает *нижний якорный узел* (lower anchor node – LAN). Это ближайший к UAN узел, для которого время ответа правого поддерева не выходит за установленные проектировщиком границы относительно времени ответа правого поддерева UAN. Интуитивно понятно, что LAN выбирается, так чтобы время ответа его правого поддерева было **близко** ко времени ответа правого поддерева соответствующего UAN. Как мы скоро увидим, это позволяет сохранять сбалансированность кустистого дерева, что сокращает время ответа.

На каждом шаге алгоритм выбирает одну из пар UAN/LAN (строго говоря, он выбирает UAN и подбирает соответствующий LAN, как описано выше) и выполняет следующее преобразование сегмента между LAN и UAN:

1) левый потомок UAN становится новым UAN преобразованного сегмента;
2) LAN остается неизменным, но его правый потомок заменяется новым узлом соединения двух поддеревьев, которые раньше были правыми поддеревьями исходных UAN и LAN.

Узел UAN, рассматриваемый на данной итерации, выбирается, исходя из следующей эвристики: выбрать UAN_B, если время ответа левого поддерева меньше, чем время ответа поддерева UAN_T, иначе выбрать UAN_T. Если время ответа одинаково, то выбрать тот узел, поддерево которого менее сбалансировано.

В конце каждого шага преобразования UAN_B и UAN_T корректируются. Алгоритм завершается, когда $UAN_B = UAN_T$, поскольку это означает, что дальнейшее преобразование невозможно. Получающееся в результате дерево выполнения соединений почти сбалансировано и дает план выполнения, время ответа которого уменьшилось благодаря параллельному выполнению соединений.

Описанный алгоритм начинается с леволинейного дерева выполнения соединений, сгенерированного оптимизатором централизованной СУБД. Такие оптимизаторы умеют генерировать очень хорошие планы, но начальный линейный план может не в полной мере учитывать особенности распределенных мультибаз, в частности репликацию данных. Специальный глобальный алгоритм оптимизации запросов может принять их во внимание. Один такой алгоритм берет начальный план и проверяет различные способы расстановки скобок в этом линейном порядке соединения, стремясь найти такой способ, при котором время ответа оптимально. Результатом является почти сбалансированное дерево выполнения соединений. Этот подход может дать более качественный план ценой увеличения времени оптимизации.

Подход на основе операторов

Выражение возможностей составляющих СУБД через реляционные операторы обеспечивает тесную интеграцию обработки запроса посредником и обертками. В частности, взаимодействие между посредником и оберткой можно выразить в терминах подпланов. Мы проиллюстрируем этот подход на примере функций планирования, предложенных в проекте Garlic. Возможности составляющих СУБД выражаются обертками в виде функций

планирования, которые могут непосредственно вызываться централизованным оптимизатором запросов. Оптимизатор, основанный на правилах, дополняется операторами для создания временных отношений и извлечения локально хранящихся данных. Также вводится оператор PushDown, который передает часть работы составляющей СУБД, которая будет ее выполнять. Планы выполнения как обычно представляются деревьями операторов, но операторные узлы аннотированы дополнительной информацией об источниках операндов, о том, где материализуются результаты, и т. д. Затем деревья операторов Garlic транслируются в операторы, которые подсистема выполнения может выполнить непосредственно.

Функции планирования рассматриваются оптимизатором как правила перечисления. Оптимизатор вызывает их для построения подпланов с помощью двух основных функций: accessPlan для доступа к отношению и joinPlan для соединения двух отношений с использованием планов доступа. Эти функции образуют единый формализм, точно отражающий возможности составляющих СУБД.

Пример 7.19. Рассмотрим три составляющие базы данных, каждая в своем узле. В базе db_1 хранится отношение EMP(ENO, ENAME, CITY), а в базе db_2 – отношение WORKS(ENO, PNAME, DUR). В базе db_3 хранится только информация о связи между работниками и проектами – в схеме с единственным отношением EMPASG(ENAME, CITY, PNAME, DUR) с первичным ключом (ENAME, PNAME). У составляющих баз данных db_1 и db_2 одна и та же обертка w_1, а у db_3 – другая обертка w_2.

Обертка w_1 предоставляет две функции планирования, типичные для реляционной СУБД. Правило

accessPlan(R: отношение, A: список атрибутов, *P*: предикат выборки) =
scan(R, A, *P*, *db*(R))

порождает оператор последовательного просмотра, который обращается к кортежам отношения R из своей составляющей базы данных *db*(R) (здесь может быть *db*(R) = db_1 или *db*(R) = db_2), применяет предикат выборки *P* и проецирует результат на список атрибутов A. Правило

joinPlan(R_1, R_2: отношения, A: список атрибутов, *P*: предикат соединения) =
join(R_1, R_2, A, *P*),
условие: $db(R_1) \neq db(R_2)$,

порождает оператор соединения, который обращается к кортежам отношений R_1 и R_2, применяет предикат соединения *P* и проецирует результат на список атрибутов A. Условие означает, что R_1 и R_2 хранятся в разных базах данных (т. е. db_1 и db_2). Таким образом, оператор соединения реализуется оберткой.

Обертка w_2 также предоставляет две функции планирования. Правило

accessPlan(R: отношение, A: список атрибутов, *P*: предикат выборки) =
fetch(CITY="c"),
условие: (CITY="c") $\subseteq$ *P*,

порождает оператор выборки, который напрямую обращается к тем кортежам работников в составляющей базе db_3, для которых CITY принимает значение "c". Правило

accessPlan(R: отношение, A: список атрибутов, P: предикат выборки) =
scan(R, A, P)

порождает оператор последовательного просмотра, который обращается к кортежам отношения R, принадлежащего обертке, применяет предикат выборки P и проецирует результат на список атрибутов A. Таким образом, оператор просмотра реализован оберткой, а не составляющей СУБД.

Рассмотрим следующий SQL-запрос, отправленный посреднику m:

```
SELECT ENAME, PNAME, DUR
FROM   EMPASG
WHERE  CITY = "Paris" AND DUR > 24
```

В предположении, что применяется подход ГКП, глобальное представление EMPASG(ENAME, CITY, PNAME, DUR) можно определить следующим образом (для простоты каждому отношению предшествует имя его составляющей базы данных):

EMPASG = (db_1.EMP ⋈ db_2.WORKS) ∪ db_3.EMPASG.

После переписывания и оптимизации запроса подход на основе операторов мог бы породить ПВЗ, показанный на рис. 7.15. Из этого плана видно, что операторы, не поддерживаемые составляющей СУБД, должны быть реализованы обертками или посредником. ♦

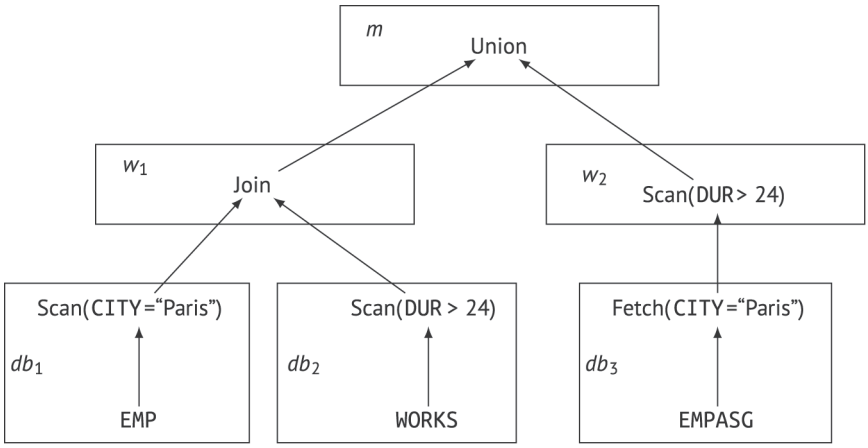


Рис. 7.15 ❖ План выполнения гетерогенного запроса

Использование функций планирования для оптимизации гетерогенных запросов в СУМБД имеет несколько преимуществ. Во-первых, функции планирования позволяют гибко и точно выразить возможности составляющих

источников данных. В частности, с их помощью можно моделировать нереляционные источники данных, например веб-сайты. Во-вторых, поскольку эти правила декларативные, разработка оберток упрощается. Трудоемкой частью разработки оберток является только реализация конкретных операторов, например оператора последовательного просмотра db_3 в примере 7.19. Наконец, этот подход легко включить в существующий централизованный оптимизатор запросов.

Подход на основе операторов был успешно применен в DIMDBS – СУМБД, предназначенной для доступа к нескольким базам данных через веб. В проекте DISCO используется ГКП-подход и поддерживается объектная модель данных для представления схем и типов данных посредника и составляющих баз данных. Это позволяет легко добавлять новые составляющие базы, без труда обрабатывая возможные несоответствия типов. Возможности составляющих СУБД определены как подмножество алгебраической подсистемы (с обычными операторами типа последовательного просмотра, соединения и объединения), которая может полностью или частично поддерживаться обертками либо посредником. Это позволяет авторам оберток гибко подходить к вопросу о том, где именно поддерживать возможности СУБД (в обертке или в посреднике). Кроме того, можно специфицировать композиции операторов, включая конкретные наборы данных, чтобы отразить ограничения составляющих СУБД. Однако использование алгебраической подсистемы и композиций операторов усложняет обработку запросов. После переписывания запроса с целью представить его как объединение подзапросов к схемам составляющих СУБД необходимо выполнить три основных шага.

1. **Генерирование пространства поиска.** Запрос разлагается на несколько ПВЗ, которые образуют пространство поиска для оптимизации запроса. Пространство поиска генерируется с применением традиционных стратегий поиска, например динамического программирования.
2. **Декомпозиция ПВЗ.** Каждый ПВЗ разлагается в лес, состоящий из *n ПВЗ-оберток* и *композиционного ПВЗ*. Каждая ПВЗ-обертка является наибольшей частью исходного ПВЗ, который может быть целиком выполнен оберткой. Операторы, которые обертка не может выполнить, поднимаются в композиционный ПВЗ. Композиционный ПВЗ объединяет результаты ПВЗ-оберток в окончательный ответ, как правило, с помощью объединений и соединений промежуточных результатов, порожденных обертками.
3. **Оценка стоимости.** Стоимость каждого ПВЗ оценивается с помощью иерархической модели стоимости, рассмотренной в разделе 7.2.4.1.

7.2.5. Трансляция и выполнение запроса

Трансляция и выполнение запроса производятся обертками в составляющих СУБД. Обертка инкапсулирует детали одной или нескольких составляющих баз данных, каждая из которых поддерживается одной и той же СУБД (или файловой системой). Кроме того, она экспортирует посреднику возможности

и функции стоимости составляющих СУБД, пользуясь единым интерфейсом. Одно из основных практических применений оберток – дать возможность СУБД на основе SQL обращаться к базам данных, в которых SQL не используется.

Главная функция обертки – преобразование между единым и СУБД-зависимым интерфейсом. На рис. 7.16 показаны различные уровни интерфейсов между посредником, оберткой и составляющими СУБД. Заметим, что в зависимости от уровня автономности составляющих СУБД эти три компонента могут находиться в разных местах. Например, в случае сильной автономности обертка должна располагаться в узле посредника, возможно, на том же сервере. Следовательно, взаимодействие между оберткой и обернутой ей составляющей СУБД включается в затраты на сетевой обмен данными. Но в случае кооперативных составляющих баз данных (например, внутри одной организации) обертку можно установить в узле СУБД, как драйвер ODBC. Тогда взаимодействие между оберткой и составляющей СУБД будет гораздо эффективнее.

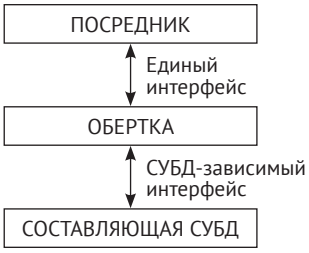


Рис. 7.16 ❖ Интерфейсы обертки

Информация, необходимая для преобразования, хранится в схеме-обертке, которая включает локальную схему, экспортированную посреднику через единый интерфейс (например, реляционную), и отображения схемы, необходимые для преобразования данных между локальной схемой и схемой составляющей базы данных, и наоборот. Отображения схем мы обсуждали в разделе 7.1.4. Требуется два вида отображений. Во-первых, обертка должна транслировать входной ПВЗ, сгенерированный посредником и выраженный в едином интерфейсе, в обращения к составляющим СУБД через СУБД-зависимый интерфейс. Эти обращения запускают выполнение запросов составляющими СУБД, которые возвращают результаты, выражаемые в СУБД-зависимом интерфейсе. Во-вторых, обертка должна транслировать эти результаты в формат единого интерфейса, чтобы их можно было вернуть посреднику для интеграции. Дополнительно обертка может выполнять операции, не поддерживаемые самой составляющей СУБД (например, операция последовательного просмотра оберткой w_2 на рис. 7.15).

Как мы говорили в разделе 7.2.4.2, единый интерфейс к оберткам может быть основан на запросах или на операторах. Задачи трансляции в обоих случаях похожи. В следующем примере для иллюстрации трансляции запроса мы выбрали подход на основе запросов в стандарте SQL/MED, который позволяет

реляционной СУБД обращаться к внешним данным, представленным в виде внешних отношений в локальной схеме обертки. Этот пример показывает, как очень простой источник данных можно обернуть для доступа через SQL.

Пример 7.20. Рассмотрим отношение EMP(ENO, ENAME, CITY), хранящееся в очень простой составляющей базе данных на сервере *ComponentDB*, состоящей из текстовых файлов в формате Unix. Каждый кортеж EMP находится в одной строке файла, атрибуты разделены знаком «:». В SQL/MED определение этого отношения в локальной схеме вместе с отображением на текстовый файл можно объявить в виде внешнего отношения с помощью такой команды:

```
CREATE  FOREIGN TABLE EMP
          ENO INTEGER,
          ENAME VARCHAR(30),
          CITY VARCHAR(20)
SERVER ComponentDB
OPTIONS (Filename '/usr/EngDB/emp.txt',
         Delimiter ':')
```

Теперь посредник может отправлять SQL-команды обертке, которая поддерживает доступ к этому отношению. Например, запрос

```
SELECT ENAME
FROM   EMP
```

может быть транслирован оберткой в следующую команду оболочки Unix, которая извлекает нужный атрибут:

```
cut -d: -f2 /usr/EngDB/emp
```

Для дополнительной обработки, например преобразования типа, можно написать программный код. ♦

Обертки чаще всего используются для запросов чтения, в этом случае трансляция запросов и построение обертки не вызывают особых трудностей. Для построения обертки обычно используются инструменты, включающие повторно используемые компоненты для генерирования большей части кода обертки. Кроме того, производители СУБД предоставляют обертки для прозрачного доступа к своим СУБД с помощью стандартных интерфейсов. Но создание оберток становится куда более сложным делом, если требуется поддержать обновление составляющих баз данных через обертку, а не прямое обновление с помощью соответствующих СУБД. Основная проблема связана с гетерогенностью ограничений целостности в едином и СУБД-зависимом интерфейсе. В главе 3 мы говорили, что ограничения целостности призваны отвергать обновления, нарушающие согласованность базы данных. В современных СУБД ограничения целостности явные и задаются в виде правил, являющихся частью схемы базы данных. Но в старых СУБД и простых источниках данных (например, файлах) ограничения целостности неявные и реализованы в прикладном коде. Например, в примере 7.20 в приложении мог бы присутствовать код, отвергающий попытки вставить в текстовый файл

EMP строку с уже существующим ENO. Этот код соответствует ограничению уникального ключа над атрибутом ENO в отношении EMP, но в обертке его нет. Таким образом, основная проблема обновления через обертку – гарантировать согласованность базы данных, отвергая любые обновления, нарушающие ограничения целостности, явные или неявные. На уровне программной инженерии эта проблема решается методами обратного конструирования, которые выявляют в коде приложения неявные ограничения целостности и преобразуют их в код проверки, входящий в состав оберток.

Еще одна серьезная проблема – сопровождение оберток. Трансляция запросов зависит от отображения между схемой составляющей базы данных и схемой обертки. Если схема составляющей базы данных со временем изменяется, то отображение может стать недействительным. Например, в примере 7.20 администратор может изменить порядок полей в файле EMP. Если используются недействительные отображения, то результаты, возвращаемые оберткой, неверны. Так как составляющие базы данных автономны, обнаружение и исправление недействительных отображений становится важной задачей. Для этого применяются методы обслуживания отображений, рассмотренные выше в этой главе.

7.3. Заключение

В данной главе мы обсудили процесс проектирования базы данных снизу вверх, который назвали интеграцией баз данных, а также вопрос о том, как выполнять запросы к базам данных, построенным таким образом. Интеграция баз данных – это процесс создания ГКС (или опосредованной схемы) и отображения на нее ЛКС. Важнейшая линия раздела проходит между хранилищами данных, в которых ГКС материализуется, и системами интеграции данных, в которых ГКС является просто виртуальным представлением.

Хотя интеграция баз данных была предметом активного изучения на протяжении многих лет, исследования сильно фрагментированы. Есть отдельные проекты, посвященные сопоставлению схем, очистке данных или отображению схем. А нужна сквозная полуавтоматическая методология интеграции баз данных с достаточным числом точек, в которых могли бы вмешаться эксперты. Один такой подход представлен в работе [Bernstein and Melnik 2007], где заложены начала полной сквозной методологии.

В литературе также уделено много внимания родственной концепции *обмена данными*, которая определяется как «задача о том, как из данных, структурированных в соответствии с исходной схемой, создать экземпляр целевой схемы, как можно точнее отражающей исходные данные» [Fagin et al. 2005]. Это очень похоже на физическую (т. е. материализованную) интеграцию данных, какая встречается в хранилищах данных. Разница между хранилищами данных и материализацией в средах обмена данными заключается в том, что хранилища обычно принадлежат одной организации и могут быть структурированы согласно четко определенной схеме, тогда как в среды обмена данными данные могут поступать из разных источников и быть гетерогенными.

В этой главе нас интересовала интеграция *баз данных*. Однако в распределенных приложениях все чаще используются данные, не хранящиеся в базах. Возникла новая интересная дисциплина – интеграция *структурированных* данных, хранящихся в базах данных, и *неструктурированных* данных из других систем (веб-серверов, мультимедийных систем, цифровых библиотек и т. д.). Мы обсудим эти вопросы в главе 12, где будем заниматься интеграцией данных из различных веб-репозиториев.

Еще один вопрос, мимо которого мы прошли в этой главе, – интеграция данных в случае, когда ГКС не существует или не может быть определена. Эта проблема особенно характерна для одноранговых систем, в которых масштаб и разнообразие источников данных столь велики, что спроектировать ГКС очень трудно (если вообще возможно). Интеграцию данных в одноранговых системах мы обсудим в главе 9.

Вторая часть этой главы посвящена проблеме обработки запросов в мультибазовых системах, которая значительно труднее, чем в тесно интегрированных однородных распределенных СУБД. Мало того что составляющие базы данных распределенные, они еще могут быть автономными, поддерживать разные языки баз данных и возможности обработки запросов и обладать различным поведением. В частности, они могут варьироваться от полноценных баз данных на основе SQL до очень простых источников (например, текстовых файлов).

В этой главе мы решали эти проблемы путем расширения и модификации архитектуры распределенной обработки запросов, описанной в главе 4. Приняв популярную архитектуру посредник–обертка, мы выделили три основных уровня, на которых запрос последовательно переписывается (и в конечном итоге адресуется к локальным отношениям) и оптимизируется посредником, а затем транслируется и выполняется обертками и составляющими СУБД. Мы также обсудили, как поддержать OLAP-запросы в мультибазах данных – важное требование к приложениям для поддержки принятия решений. Для этого необходим дополнительный уровень трансляции многомерных OLAP-запросов в реляционные. Эта многоуровневая архитектура обработки мультибазовых запросов настолько общая, что в нее укладываются самые разные вариации. Она с успехом применялась для описания разнообразных методов обработки запросов, спроектированных с разными целями и в разных предположениях.

Основные способы обработки мультибазовых запросов – переписывание запросов с применением мультибазовых представлений, оптимизации и выполнения мультибазовых запросов, а также трансляция и выполнение запросов. Методы на основе мультибазовых представлений существенно зависят от того, какой подход к интеграции используется: ГКП или ЛКП. Переписывание запросов в случае ГКП напоминает локализацию данных в однородных распределенных системах баз данных. Методы же для ЛКП (и его обобщения ГЛКП) гораздо сложнее; зачастую не удается найти эквивалентного переписывания запроса, и тогда необходим запрос, порождающий максимальное подмножество истинного ответа. Методы оптимизации мультибазовых запросов включают моделирование стоимости и оптимизацию запроса для составляющих баз данных с разными вычислительными возможностями.

Эти методы расширяют традиционную обработку распределенных запросов, делая акцент на гетерогенности. Помимо гетерогенности, есть еще одна важная проблема – учесть динамическое поведение составляющих СУБД. Техника адаптивной обработки запросов решает эту проблему за счет того, что оптимизатор запросов во время выполнения взаимодействует с исполняющей средой и реагирует на непредвиденные изменения условий выполнения. Наконец, мы обсудили методы трансляции запросов для выполнения составляющими СУБД и для генерирования и управления обертками.

Модель данных, используемая посредником, может быть реляционной, объектно-ориентированной или еще какой-то. В этой главе мы для простоты предполагали, что посредник работает с реляционной моделью, этого достаточно, чтобы объяснить методы обработки мультибазовых запросов. Но при работе с источниками данных в вебе может понадобиться более развитая модель посредника, например объектно-ориентированная или слабо структурированная (скажем, основанная на XML или RDF). Для этого потребуется существенно расширить методы обработки запросов.

7.4. Библиографические замечания

На тему этой главы существует обширная литература. Многие исследования были выполнены еще в начале 1980-х годов, их хороший обзор имеется в работе [Batini et al. 1986]. К более поздним относятся работы [Elmagarmid et al. 1999] и [Sheth and Larson 1990]. Еще один неплохой обзор этой области – работа [Jhingran et al. 2002].

В книге [Doan et al. 2012] эта тема обсуждается во всей полноте. Существует также немало обзорных статей. В работе [Bernstein and Melnik 2007] имеется отличное обсуждение методологии интеграции, а также сравнение работ по управлению моделями и исследований по интеграции данных. Статья [Halevy et al. 2006] представляет собой обзор работ по интеграции данных, проделанных в 1990-е годы с упором на систему Information Manifold [Levy et al. 1996a], в которой используется подход ЛКП. Приводится обширная библиография и обсуждаются области исследования, появившиеся за прошедшие с тех пор годы. В работе [Haas 2007] принят всеобъемлющий подход к процессу интеграции в целом, который разбит на четыре этапа: выявление релевантной информации (ключей, ограничений, типов данных и т. д.) и ее анализ с целью оценить качество данных и определить их статистические свойства; стандартизация, в ходе которой определяется наилучший способ представить интегрированные данные; спецификация, т. е. конфигурирование процесса интеграции, и выполнение, когда, собственно, и производится интеграция. На этапе спецификации применяются методы, определенные в той же статье.

Подходы ЛКП и ГКП определены и рассмотрены в работах [Lenzerini 2002], [Koch 2001] и [Calì and Calvanese 2002]. Подход ГЛКП обсуждается в работах [Friedman et al. 1999] и [Halevy 2001]. Для сравнения ЛКП и ГКП разработан целый ряд систем. Во многих из них упор сделан на запросы к интегриро-

ванным системам. Примеры ЛКП-подхода описаны в работах [Duschka and Genesereth 1997, Levy et al. 1996b, Manolescu et al. 2001], а применение ГКП – в работах [Adali et al. 1996a, Garcia-Molina et al. 1997, Haas et al. 1997b].

Вопросы структурной и семантической гетерогенности занимают исследователей довольно давно. Литература на эту тему весьма обширна, поэтому отметим лишь несколько интересных публикаций по структурной гетерогенности – [Dayal and Hwang 1984, Kim and Seo 1991, Breitbart et al. 1986, Krishnamurthy et al. 1991, Batini et al. 1986] (в работе [Batini et al. 1986] обсуждаются также структурные конфликты, кратко описанные в этой главе) – и по семантической гетерогенности – [Sheth and Kashyap 1992, Hull 1997, Ouksel and Sheth 1999, Kashyap and Sheth 1996, Bright et al. 1994, Ceri and Widom 1993, Vermeer 1997]. Отметим, что этот список далеко не полон.

Имеется несколько предложений канонической модели для ГКС. Перечислим только те, что обсуждались в этой главе: модель сущность–связь [Palopoli et al. 1998, Palopoli 2003, He and Ling 2006], объектно-ориентированная модель [Castano and Antonellis 1999, Bergamaschi 2001], графовая модель (она используется также для определения структурного сходства) [Palopoli et al. 1999, Milo and Zohar 1998, Melnik et al. 2002, Do and Rahm 2002, Madhavan et al. 2001], древесная модель [Madhavan et al. 2001] и XML [Yang et al. 2003].

В работе [Doan and Halevy 2005] имеется прекрасный обзор методов сопоставления схем и предлагается другая, более простая классификация методов: на основе правил, на основе обучения и комбинированные. Другие работы по сопоставлению схем описаны в обзоре [Rahm and Bernstein 2001], где можно найти отличное сравнение различных предложений. Межсхемные правила, которые мы обсуждали в этой главе, взяты из работы [Palopoli et al. 1999]. Классический источник по функциям ранжирующего агрегирования, используемого при сопоставлении, – работа [Fagin 2002].

Несколько систем разработано для демонстрации практической применимости разных подходов к сопоставлению схем. Из методов на основе правил отметим системы DIKE [Palopoli et al. 1998, Palopoli 2003, Palopoli et al. 2003], DIPE – более ранняя версия этой системы [Palopoli et al. 1999], TranSCM [Milo and Zohar 1998], ARTEMIS [Bergamaschi 2001], распространение сходства [Melnik et al. 2002], CUPID [Madhavan et al. 2001] и COMA [Do and Rahm 2002]. Что касается сопоставления на основе обучения, упомянем систему Autoplex [Berlin and Motro 2001], в которой реализован наивный байесовский классификатор; этот же подход предложен в работах [Doan et al. 2001, 2003a] и [Naumann et al. 2002]. В эту же категорию попадают решающие деревья, обсуждаемые в работе [Embley et al. 2001, 2002], а также система iMAP [Dhamankar et al. 2004].

В работах [Roth and Schwartz 1997, Tomasic et al. 1997, Thiran et al. 2006] изучаются различные аспекты технологии оберток. Программное решение проблемы создания и сопровождения оберток с учетом контроля целостности предложено в работе [Thiran et al. 2006].

Из работ по бинарной интеграции упомянем [Batini et al. 1986, Pu 1988, Batini and Lenzirini 1984, Dayal and Hwang 1984, Melnik et al. 2002], а n-арные механизмы обсуждаются в работах [Elmasri et al. 1987, Yao et al. 1982, He et al. 2004]. О некоторых инструментах интеграции баз данных можно прочитать

в работах [Sheth et al. 1988a], [Miller et al. 2001], где обсуждается программа Clio, и [Roitman and Gal 2006], где описывается OntoBuilder.

Алгоритм создания отображений из раздела 7.1.4.1 предложен в работах [Miller et al. 2000], [Yan et al. 2001] и [Popa et al. 2002]. Обслуживание отображений обсуждается в работе [Velegrakis et al. 2004].

Тема очистки данных привлекла значительный интерес в последние годы, когда усилия по интеграции открыли новые источники данных для более широкого использования. Литературы на эту тему много, к тому же она подробно освещается в книге [Ilyas and Chu 2019]. Различие между очисткой на уровне схемы и на уровне экземпляра сформулировано в работе [Rahm and Do 2000]. Мы обсуждали следующие операторы очистки данных: расщепление столбцов [Raman and Hellerstein 2001], оператор отображения [Galhardas et al. 2001] и нечеткое сопоставление [Chaudhuri et al. 2003].

Первые исследования по обработке мультибазовых запросов относятся к началу 1980-х годов, когда появились системы управления мультибазами данных (например, [Brill et al. 1984, Dayal and Hwang 1984] и [Landers and Rosenberg 1982]). Ставилась цель обеспечить доступ к разным базам данных внутри организации. В 1990-х годах возрастающее применение веба для доступа к самым разным источникам данных возродило интерес к обработке мультибазовых запросов, и появилось много новых работ в русле популярной архитектуры посредник–обертка [Wiederhold 1992]. Краткий обзор работ по оптимизации мультибазовых запросов можно найти в статье [Meng et al. 1993]. Хорошие обсуждения обработки мультибазовых запросов имеются в работах [Lu et al. 1992, 1993], в главе 4 книги [Yu and Meng 1998] и в статье [Kossmann 2000].

Переписывание запросов с помощью представлений обсуждается в статье [Levy et al. 1995], а обзор работ на эту тему см. в [Halevy 2001]. В работе [Levy et al. 1995] показано, что общая задача нахождения переписывания с помощью представлений является NP-полной относительно количества представлений и числа подцелей в запросе. Техника разворачивания для переписывания запроса, выраженного на языке Datalog в рамках подхода ГКП, предложена в работе [Ullman 1997]. Основные методы переписывания запроса в подходе ЛКП – алгоритм корзин [Levy et al. 1996b], алгоритм обратного правила [Duschka and Genesereth 1997] и алгоритм MinCon [Pottinger and Levy 2000].

Три основных подхода к гетерогенному моделированию стоимости обсуждаются в работе [Zhu and Larson 1998]. Подход на основе черного ящика используется в работах [Du et al. 1992, Zhu and Larson 1994]; в эту группу входят следующие методы: пробные запросы [Zhu and Larson 1996a], выборочные запросы (частный случай пробных) [Zhu and Larson 1998] и обучение по мере предъявления и ответов на запросы [Adali et al. 1996b]. Специализированный подход развит в работах [Zhu and Larson 1996b, Roth et al. 1999, Naacke et al. 1999]; в частности, вычислять стоимость можно внутри обертки (как в Garlic) [Roth et al. 1999] или с помощью иерархической модели стоимости (как в Disco) [Naacke et al. 1999]. Динамический подход применен в работах [Zhu et al. 2000], [Zhu et al. 2003] и [Rahal et al. 2004] и обсуждается также в работе [Lu et al. 1992]. В работе [Zhu 1995] обсуждается динамическая выборка, а в работе [Zhu et al. 2000] представлен качественный подход.

Алгоритм, который мы описали для иллюстрации основанного на запросах подхода к оптимизации гетерогенных запросов (раздел 7.2.4.2), предложен в работе [Du et al. 1995] и обсуждается в работе [Evrendilek et al. 1997]. Для иллюстрации подхода, основанного на операторах, мы воспользовались популярным решением с использованием функций планирования, предложенным в проекте Garlic [Haas et al. 1997a]. Подход на основе операторов использовался также в DISCO – мультибазовой системе для доступа к составляющим базам данных через веб [Tomasic et al. 1996, 1998].

Адаптивная обработка запросов изучалась многими исследователями в разных средах. В работе [Amsaleg et al. 1996] показано, почему статические планы не могут справиться с непредсказуемостью источников данных; существуют проблемы с постоянными запросами [Madden et al. 2002b], дорогостоящими предикатами [Porto et al. 2003] и асимметрией данных [Shah et al. 2003]. Обзоры работ по адаптивным методам – [Hellerstein et al. 2000, Gounaris et al. 2002]. Самый известный динамический подход – вихревой оператор (см. главу 4), обсуждаемый в работе [Avnur and Hellerstein 2000]. Другие важные методы адаптивной обработки запросов – реорганизация запроса [Amsaleg et al. 1996, Urhan et al. 1998], пульсирующие соединения [Haas and Hellerstein 1999b], адаптивное секционирование [Shah et al. 2003] и избирательный подход [Porto et al. 2003]. Основными расширениями вихря являются модули состояний [Raman et al. 2003] и распределенные вихри [Tian and DeWitt 2003].

В этой главе в центре нашего внимания была интеграция структурированных данных, хранящихся в базах данных. Более общая задача интеграции структурированных и неструктурированных данных обсуждается в работах [Halevy et al. 2003] и [Somani et al. 2002]. Другое направление обобщения исследовано в работе [Bernstein and Melnik 2007], где предложен механизм управления моделями, который «поддерживает операции сопоставления схем, композиции отображений, вычисления разности схем, объединения схем, трансляции схем в другие модели данных и генерирования преобразований данных по отображениям».

Помимо отмеченных выше систем, в этой главе упоминался ряд других. Перечислим их вместе с соответствующими источниками: SEMINT [Li and Clifton 2000, Li et al. 2000], ToMAS [Velegrakis et al. 2004], Maveric [McCann et al. 2005] и Aurora [Yan 1997, Yan et al. 1997].

Упражнения

Задача 7.1. Распределенные системы баз данных и распределенные системы мультибаз данных – два разных подхода к проектированию систем. Найдите три практических приложения, для которых лучше подходит тот или иной подход. Обсудите, в силу каких особенностей приложений один подход оказывается предпочтительнее другого.

Задача 7.2. В одних архитектурных моделях определение глобальной концептуальной схемы считается полезным, в других – нет. Как вы думаете, по-

чему? Приведите подробные технические аргументы в обоснование своего ответа.

Задача 7.3 (*). Придумайте алгоритм преобразования реляционной схемы в модель сущность–связь.

Задача 7.4 (**). Рассмотрите две базы данных на рис. 7.17 и 7.18, описания которых приведены ниже. Спроектируйте глобальную концептуальную схему в виде объединения обеих баз данных, предварительно преобразовав их в модель сущность–связь.

На рис. 7.17 описана реляционная база данных, используемая организаторами автомобильных гонок, а на рис. 7.18 – модель сущность–связь для базы данных, используемой производителем обуви. Ниже обсуждается семантика обеих схем.

```
DIRECTOR(NAME, PHONE_NO, ADDRESS)
LICENSES(LIC_NO, CITY, DATE, ISSUES, COST, DEPT, CONTACT)
RACER(NAME, ADDRESS, MEM_NUM)
SPONSOR(SP_NAME, CONTACT)
RACE(R_NO, LIC_NO, DIR, MAL_WIN, FRM_WIN, SP_NAME)
```

Рис. 7.17 ❖ База данных об автомобильных гонках

DIRECTOR – отношение, в котором определены директоры, отвечающие за организацию гонок; предполагается, что у каждого директора имеется уникальное имя (используется как первичный ключ), номер телефона и адрес.

LICENSES – необходимо, поскольку у любой гонки должна быть выданная государством лицензия, которая выпускается контактным лицом CONTACT в отделе ISSUER, который, возможно, является частью министерства DEPT; у лицензии имеется уникальный номер LIC_NO (ключ), который действителен для проведения гонок в городе CITY на дату DATE и имеет стоимость COST.

RACER – это отношение описывает участников гонки. Каждый участник имеет имя NAME, которого недостаточно для уникальной идентификации, поэтому для образования составного ключа к нему присоединяется адрес ADDRESS. Наконец, каждому участнику может быть присвоен членский номер MEM_NUM, который идентифицирует его как члена братства гонщиков, но не все участники имеют такие номера.

SPONSOR – описывает, кто спонсирует гонку. Часто бывает, что одна организация спонсирует несколько гонок, и для контактов назначается конкретное лицо (CONTACT). У разных гонок могут быть разные спонсоры.

RACE – однозначно идентифицирует гонку с номером лицензии LIC_NO по номеру гонки (R_NO) (этот атрибут используется в качестве ключа, потому что гонку можно запланировать, еще не получив лицензии); в каждой гонке есть победитель в мужском и женском разряде (MAL_WIN и FEM_WIN) и директор гонки (DIR).

На рис. 7.18 показана модель сущность–связь, используемая в системе производителя обуви. Ниже описана ее семантика.

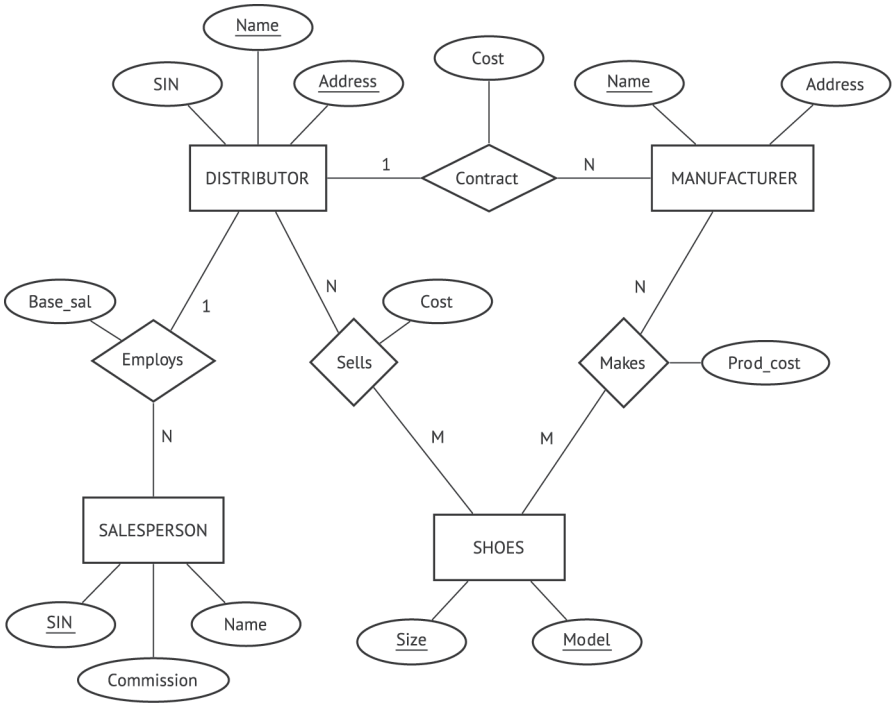


Рис. 7.18 ❖ База данных производителя обуви

SHOES – обувь, характеризуемая моделью MODEL и размером SIZE, которые вместе образуют ключ сущности.

MANUFACTURER – производитель, однозначно идентифицируется ключом NAME и находится по адресу ADDRESS.

DISTRIBUTOR – дистрибьютер, юридическое лицо, имеющее название NAME и адрес ADDRESS (в совокупности образующие ключ) и идентификатор SIN для налогообложения.

SALESPERSON – продавец, юридическое лицо, имеющее название NAME, которое получает комиссионные COMMISSION и однозначно идентифицируется своим налоговым номером SIN (ключом).

Makes – связь «изготавливает», с которой ассоциирована фиксированная стоимость производства (PROD_COST). Показывает, что некий производитель изготавливает различную обувь и что разные производители могут изготавливать одну и ту же обувь.

Sells – связь «продает», с которой ассоциирована оптовая цена COST для дистрибьютера обуви. Показывает, что каждый дистрибьютер продает обувь многих типов и что обувь каждого типа продается многими дистрибьютерами.

Contract – связь, показывающая, что дистрибьютер приобретает за сумму COST эксклюзивное право представлять изготовителя. Заметим, что при этом дистрибьютеру не запрещено продавать обувь других производителей.

Employs – показывает, что каждый дистрибьютер нанимает несколько продавцов, которые продают обувь; каждый из них получает зарплату BASE_SALARY.

Задача 7.5 (*). Рассмотрим три источника:

- база данных 1 содержит отношение `Area(Id, Field)`, описывающее области специализации работников, поле `Id` идентифицирует работника;
- база данных 2 содержит два отношения: `Teach(Professor, Course)` и `In(Course, Field)`; `Teach` показывает, какие курсы преподает профессор, а `In` – области знаний, к которым может относиться курс;
- база данных 2 содержит два отношения: `Grant(Researcher, GrantNo)` описывает гранты, полученные исследователями, а `For(GrantNo, Field)` – на какие области исследования выделены гранты.

Задача заключается в том, чтобы построить ГКС с двумя отношениями: `Works(Id, Project)` показывает, что работник занимается конкретным проектом, а `Area(Project, Field)` ассоциирует проект с одной или несколькими областями знаний.

a) Предъявите ЛКП-отображение между базой данных 1 и ГКС.
b) Предъявите ГЛКП-отображение между ГКС и локальными схемами.
c) Предположим, что в базу данных 3 добавлено еще одно отношение, `Funds(GrantNo, Project)`. Предъявите ГКП-отображение в этом случае.

Задача 7.6. Рассмотрим ГКС со следующим отношением: `Person(Name, Age, Gender)`. Это отношение определено как представление над тремя ЛКС:

```
CREATE VIEW Person AS
SELECT Name, Age, "male" AS Gender
FROM   SoccerPlayer
UNION
SELECT Name, NULL AS Age, Gender
FROM   Actor
UNION
SELECT Name, Age, Gender
FROM   Politician
WHERE  Age > 30
```

Для каждого из следующих запросов ответьте, какая из трех локальных схем (`SoccerPlayer`, `Actor`, `Politician`) дает вклад в результат глобального запроса.

a) `SELECT Name FROM Person`
b) `SELECT Name FROM Person WHERE Gender = "female"`
c) `SELECT Name FROM Person WHERE Age > 25`
d) `SELECT Name FROM Person WHERE Age < 25`
e) `SELECT Name FROM Person WHERE Gender = "male" AND Age = 40`

Задача 7.7. ГКС, содержащая отношение `Country(Name, Continent, Population, HasCoast)`, описывает страны мира. Атрибут `HasCoast` показывает, имеет ли страна выход к морю. С глобальной схемой связаны три ЛКС с использованием подхода ЛКП:

```
CREATE VIEW EuropeanCountry AS
SELECT Name, Continent, Population, HasCoast
FROM   Country
WHERE  Continent = "Europe"
```

```
CREATE VIEW BigCountry AS
SELECT Name, Continent, Population, HasCoast
FROM   Country
WHERE  Population >= 30000000

CREATE VIEW MidsizeOceanCountry AS
SELECT Name, Continent, Population, HasCoast
FROM   Country
WHERE  HasCoast = true AND Population > 10000000
```

a) Для каждого из следующих запросов обсудите результаты с точки зрения их полноты, т. е. проверьте, покрывают ли локальные источники (или их комбинация) все релевантные результаты.
   1. `SELECT Name FROM Country`
   2. `SELECT Name FROM Country WHERE Population > 40`
   3. `SELECT Name FROM Country WHERE Population > 20`

b) Для каждого из следующих запросов обсудите, какие из трех ЛКС необходимы для получения результата глобального запроса.
   1. `SELECT Name FROM Country`
   2. `SELECT Name FROM Country WHERE Population > 30 AND Continent = "Europe"`
   3. `SELECT Name FROM Country WHERE Population < 30`
   4. `SELECT Name FROM Country WHERE Population > 30 AND HasCoast = true`

Задача 7.8. Рассмотрим отношения `PRODUCT` и `ARTICLE`, описанные в упрощенной нотации SQL. Точные соответствия в обеих схемах обозначены стрелками.

```
PRODUCT                       → ARTICLE
  Id: int PRIMARY KEY         → Key: varchar(255) PRIMARY KEY
  Name: varchar(255)          → Title: varchar(255)
  DeliveryPrice: float        → Price: real
  Description: varchar(8000)  → Information: varchar(5000)
```

a) Для каждого из пяти соответствий укажите, какой из следующих методов сопоставления схем, вероятно, найдет соответствие:
   1) синтаксическое сравнение имен элементов, например вычисление сходства строк в смысле редакционного расстояния;
   2) сравнение имен элементов с помощью справочной таблицы синонимов;
   3) сравнение типов данных;
   4) анализ примеров данных.

b) Позволяют ли вышеперечисленные методы сопоставления выявить ложные соответствия в этих задачах? Если да, приведите пример.

Задача 7.9. Рассмотрим два отношения `S(a, b, c)` и `T(d, e, f)`. Некий метод сопоставления определил следующие степени сходства элементов `S` и `T`:

	`T.d`	`T.e`	`T.f`
`S.a`	0.8	0.3	0.1
`S.b`	0.5	0.2	0.9
`S.c`	0.4	0.7	0.8

Исходя из этих результатов сопоставителя, выведите результат сопоставления схем в целом, обладающий следующими характеристиками:

- каждый элемент участвует ровно в одном соответствии;
- не существует такого соответствия, в котором оба элемента сопоставлены с элементом противоположной схемы с большей степенью сходства, чем в данном сопоставлении.

Задача 7.10 (*). На рис. 7.19 показаны схемы трех источников данных:

- MyGroup содержит публикации, написанные членами рабочей группы;
- MyConference содержит публикации, представленные на конференции и связанных с ней семинарах;
- MyPublisher содержит статьи, опубликованные в журналах.

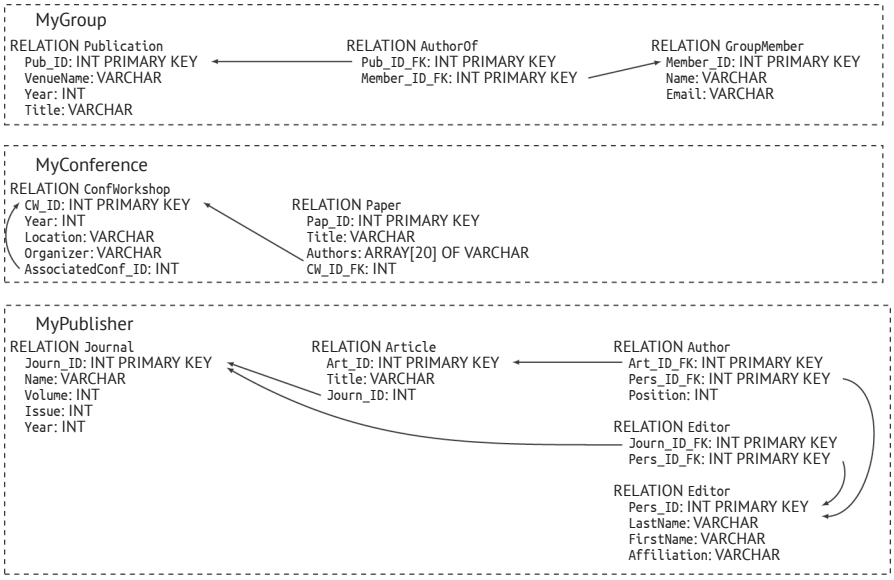


Рис. 7.19 ❖ Схемы к задаче 7.10

Стрелки показывают связи между внешним и первичным ключами; заметим, что для экономии места мы опустили правильный синтаксис SQL для задания связей внешнего ключа, заменив его стрелками.

Источники определены следующим образом:

MyGroup

- `Publication`
  - `Pub_ID`: уникальный идентификатор публикации
  - `VenueName`: название журнала, конференции или семинара
  - `VenueType`: «journal», «conference» или «workshop»
  - `Year`: год публикации
  - `Title`: название публикации
- `AuthorOf`
  - связь многие-ко-многим с семантикой «член группы является автором публикации»

- `GroupMember`
  - `Member_ID`: уникальный идентификатор члена группы
  - `Name`: имя члена группы
  - `Email`: адрес электронной почты члена группы

MyConference

- `ConfWorkshop`
  - `CW_ID`: уникальный идентификатор конференции или семинара
  - `Name`: название конференции или семинара
  - `Year`: год проведения мероприятия
  - `Location`: место проведения мероприятия
  - `Organizer`: имя организатора
  - `AssociatedConf_ID_FK`: значение равно NULL, если это конференция, и идентификатору ассоциированной конференции, если это семинар (предполагается, что любой семинар ассоциирован с какой-то конференцией)
- `Paper`
  - `Pub_ID`: уникальный идентификатор публикации
  - `Pap_ID`: уникальный идентификатор работы
  - `Title`: название работы
  - `Author`: массив имен авторов
  - `CW_ID_FK`: конференция, в материалах которой опубликована работа

MyPublisher

- `Journal`
  - `Journ_ID`: уникальный идентификатор журнала
  - `Name`: название журнала
  - `Year`: год выхода журнала
  - `Volume`: номер тома
  - `Issue`: номер выпуска
- `Article`
  - `Art_ID`: уникальный идентификатор статьи
  - `Title`: название статьи
  - `Journ_ID_FK`: журнал, в котором опубликована статья
- `Person`
  - `Pers_ID`: уникальный идентификатор человека
  - `LastName`: фамилия человека
  - `FirstName`: имя человека
  - `Affiliation`: место работы человека (например, название университета)
- `Author`
  - представляет связь многие-ко-многим с семантикой «человек является автором статьи»
  - `Position`: позиция автора в списке авторов (например, первый автор имеет позицию 1)
- `Editor`
  - представляет связь многие-ко-многим с семантикой «человек является редактором выпуска журнала».

a) Найдите все соответствия между элементами схем двух источников. Используйте имена и типы данных элементов, а также приведенное выше описание.
b) Классифицируйте соответствия по следующим критериям:
   1) тип элементов схемы (например, атрибут–атрибут или атрибут–отношение);
   2) степень (например, 1:1 ил 1:N).
c) Приведите консолидированную глобальную схему, охватывающую всю информацию из исходных схем.

Задача 7.11 (*). На рис. 7.20 показаны (в упрощенном синтаксисе SQL) два источника $Source_1$ и $Source_2$. В $Source_1$ имеется два отношения: `Course` и `Tutor`, а в $Source_2$ одно отношение: `Lecture`. Сплошными стрелками обозначены соответствия между элементами схем. Штриховая стрелка представляет связь внешнего ключа между двумя отношениями $Source_1$.

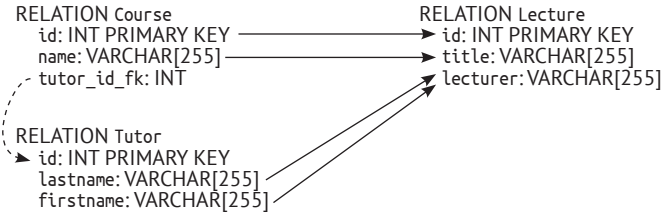


Рис. 7.20 ❖ Схемы к задаче 7.11

Ниже приведены четыре отображения схем (представленные в виде SQL-запросов) для преобразования данных $Source_1$ в $Source_2$.

1.
```
SELECT C.id, C.name as Title, CONCAT(T.lastname, T.firstname) AS Lecturer
FROM   Course AS C
JOIN   Tutor AS T ON (C.tutor_id_fk = T.id)
```

2.
```
SELECT C.id, C.name AS Title, NULL AS Lecturer
FROM   Course AS C
UNION
SELECT T.id AS ID, NULL AS Title, T, lastname AS Lecturer
FROM   Course AS C
FULL OUTER JOIN Tutor AS T ON (C.tutor_id_fk=T.id)
```

3.
```
SELECT C.id, C.name as Title, CONCAT(T.lastname, T.firstname) AS Lecturer
FROM   Course AS C
FULL OUTER JOIN Tutor AS T ON(C.tutor_id_fk=T.id)
```

Для каждого отображения ответьте на следующие вопросы:
a) Является ли отображение осмысленным?
b) Является ли отображение полным (т. е. все ли данные $Source_1$ преобразуются)?
c) Может ли отображение нарушить ограничения ключа?

Задача 7.12 (*). Рассмотрим три источника данных.

- База данных 1 состоит из одного отношения AREA(ID, FIELD), описывающего четыре области специализации работников, где ID – идентификатор работника.
- База данных 2 состоит из двух отношений: TEACH(PROFESSOR, COURSE), описывающего, какие курсы читает каждый профессор, и IN(COURSE, FIELD), описывающего области знаний, к которым относится курс.
- База данных 3 состоит из двух отношений: GRANT(RESEARCHER, GRANT#), описывающего гранты, полученные исследователями, и FOR(GRANT#, FIELD), описывающего, на какие области исследования получен грант.

Спроектируйте глобальную схему с двумя отношениями: в WORKS(ID, PROJECT) в хранится информация о проектах, в которых заняты работники, а AREA(PROJECT, FIELD) ассоциирует каждый проект с одной или несколькими областями знаний – для следующих случаев:

a) должно существовать ЛКП-отображение между базой данных 1 и глобальной схемой;

b) должно существовать ГЛКП-отображение между глобальной схемой и локальными схемами;

c) должно существовать ГКП-отображение, когда в базу данных 3 добавлено еще одно отношение FUNDS(GRANT#, PROJECT).

Задача 7.13 (**). Было предложено использовать логику (если быть точным, логику первого порядка) в качестве единообразного формализма для трансляции и интеграции схем. Обсудите, каким образом логика может быть полезна для этой цели.

Задача 7.14 (**). Можно ли произвести какую-либо глобальную оптимизацию глобальных запросов в мультибазовой системе? Формально опишите условия, при которых такая оптимизация была бы возможна.

Задача 7.15 (**). Рассмотрим глобальные отношения EMP(ENAME, TITLE, CITY) и ASG(ENAME, PNAME, CITY, DUR). Атрибут CITY в ASG – местоположение проекта с именем PNAME (т. е. PNAME функционально определяет CITY). Рассмотрим локальные отношения EMP1(ENAME,TITLE, CITY), EMP2(ENAME, TITLE, CITY), PROJ1 (PNAME, CITY), PROJ2(PNAME, CITY) и ASG1(ENAME, PNAME, DUR). Рассмотрим запрос *Q*, который выбирает имена и продолжительность занятости работников, занятых в проекте в Рио-де-Жанейро в течение более 6 месяцев.

a) В предположении, что применяется подход ГКП, выполните переписывание запроса.

b) В предположении, что применяется подход ЛКП, выполните переписывание запроса с помощью алгоритма корзин.

c) То же самое, что (b), но с помощью алгоритма MinCon.

Задача 7.16 (*). Рассмотрим отношения EMP и ASG из примера 7.18. Обозначим |R| количество страниц, необходимых для хранения R на диске. Рассмотрим следующую статистику данных:

|EMP| = 100;
|ASG| = 2000;
selectivity(ASG.DUR > 36) = 1 %.

Обобщенная модель стоимости посредника имеет вид

$cost(\sigma_{A=v}(R)) = |R|$;

$cost(\sigma(X)) = cost(X)$, где X содержит хотя бы один оператор;

$cost(R \bowtie_A^{ind} S) = cost(R) + |R| * cost(\sigma_{A=v}(S))$ при использовании соединения по индексу;

$cost(R \bowtie_A^{nl} S) = cost(R) + |R| * cost(S)$ при использовании соединения методом вложенных циклов.

Рассмотрим входной запрос Q к СУМБД:

```
SELECT *
FROM   EMP NATURAL JOIN ASG
WHERE ASG.DUR>36
```

Рассмотрим четыре плана обработки Q:

$P_1 = EMP \bowtie_{ENO}^{ind} \sigma_{DUR>36}(ASG)$;
$P_2 = EMP \bowtie_{ENO}^{nl} \sigma_{DUR>36}(ASG)$;
$P_3 = \sigma_{DUR>36}(ASG) \bowtie_{ENO}^{ind} EMP$;
$P_4 = \sigma_{DUR>36}(ASG) \bowtie_{ENO}^{nl} EMP$.

a) Какова стоимость всех четырех планов?
b) У какого плана стоимость минимальна?

Задача 7.17 (*). Рассмотрим отношения EMP и ASG из предыдущего упражнения. Предположим, что модель стоимости посредника дополнена следующей информацией о стоимости, относящейся к составляющим СУБД.

Стоимость доступа к кортежам EMP в базе db_1 равна

$cost(\sigma_{A=v}(R)) = |\sigma_{A=v}(R)|$.

Стоимость выборки из ASG кортежей с заданным значением ENO в базе db_2 равна

$cost(\sigma_{ENO=v}(ASG)) = |\sigma_{ENO=v}(ASG)|$.

a) Какова теперь стоимость всех четырех планов?
b) У какого плана стоимость минимальна?

Задача 7.18 (**). Каковы достоинства и ограничения подходов на основе запросов и операторов к оптимизации гетерогенных запросов с точки зрения выразительности запроса, производительности запроса, стоимости разработки оберток, сопровождения системы (посредника и оберток) и эволюции?

Задача 7.19 (**). Рассмотрим пример 7.19 и добавим еще один узел, в котором размещена база данных db_4, содержащая отношения EMP(ENO, ENAME, CITY) и ASG(ENO, PNAME, DUR). db_4 с помощью своей обертки w_3 экспортирует возмож-

ности соединения и последовательного просмотра. Предположим, что в db_1 могут существовать работники, для которых связи с проектами хранятся в *db*4, а в *db*4 могут существовать работники, для которых связи с проектами хранятся в *db*2.

a) Определите функции планирования обертки w_3.

b) Дайте новое определение глобального представления `EMPASG(ENAME, CITY, PNAME, DUR)`.

c) Приведите ПВЗ для того же запроса, что в примере 7.19.

Глава 8

Параллельные системы баз данных

Многим приложениям требуются базы данных гигантского объема (порядка сотен или тысяч терабайт). Эффективную поддержку очень больших баз данных в режиме OLTP или OLAP можно организовать, сочетая параллельные вычисления с управлением распределенными базами данных.

Параллельный компьютер, или мультипроцессор, – это разновидность распределенной системы, состоящая из некоторого количества узлов (процессоров, запоминающих устройств и дисков), соединенных между собой сверхбыстрой сетью и расположенных в одной или нескольких стойках в пределах одного помещения. В зависимости от степени связанности различают два вида мультипроцессоров: сильно связанные и слабо связанные. Сильно связанные мультипроцессоры содержат несколько процессоров, соединенных шиной с общей памятью. В мейнфреймах, суперкомпьютерах и современных многоядерных процессорах для повышения производительности используется сильная связанность. Слабо связанные мультипроцессоры, которые теперь называют компьютерными кластерами, или просто кластерами, состоят из нескольких серийных компьютеров, связанных высокоскоростной сетью. Основная идея заключается в том, чтобы построить мощный компьютер из большого числа небольших узлов с отличным отношением цена/производительность, который будет стоить значительно дешевле, чем эквивалентный мейнфрейм или суперкомпьютер. В простейшем и самом дешевом варианте для соединения используется обычная локальная сеть. Однако сейчас для кластеров доступны быстрые стандартные межсоединения (например, Infiniband и Myrinet), которые могут похвастаться высокой полосой пропускания (например, 100 Гбит/с) и низкой задержкой сообщений.

В предыдущих главах мы уже говорили, что распределение данных можно использовать для повышения производительности (с помощью распараллеливания) и доступности (с помощью репликации). Этот принцип можно применить для реализации *параллельных систем баз данных*, т. е. систем баз данных на параллельных компьютерах. В параллельных системах баз данных управление базами можно распараллелить для построения высокопроизводительных и высокодоступных серверов баз данных, которые смогут поддержать очень большие базы с очень высокой нагрузкой.

Большая часть исследований по параллельным системам баз данных выполнена в контексте реляционной модели, потому что она представляет хорошую основу для параллельной обработки данных. В этой главе мы опишем подход на основе параллельной системы баз данных для решения задачи о высокопроизводительном и высокодоступном управлении данными. Мы обсудим плюсы и минусы различных архитектур параллельных систем и представим общие методы реализации.

Реализация параллельных систем баз данных естественным образом опирается на методы распределенных баз данных. Однако особую важность приобретают вопросы размещения данных, параллельной обработки запросов и балансирования нагрузки, потому что количество узлов может быть гораздо больше, чем в распределенной СУБД. Кроме того, параллельный компьютер обычно обеспечивает надежную быструю связь, чем можно воспользоваться для эффективной реализации распределенного управления транзакциями и репликации. Поэтому хотя базовые принципы такие же, как в распределенных СУБД, методы работы с параллельными системами баз данных довольно сильно отличаются.

Эта глава организована следующим образом. В разделе 8.1 уточняются цели, стоящие перед параллельными системами баз данных. В разделе 8.2 мы обсудим архитектуры, в т. ч. с общей памятью, с общим диском и без разделения ресурсов. Затем в разделе 8.3 мы опишем методы размещения данных, в разделе 8.4 – обработку запросов, в разделе 8.5 – балансировку нагрузки, а в разделе 8.6 – отказоустойчивость. В разделе 8.7 будет представлено использование методов параллельного управления данными в кластерах баз данных – важной разновидности параллельных систем баз данных.

8.1. Цели

В параллельной обработке многопроцессорные компьютеры применяются для выполнения прикладных программ сразу несколькими совместно работающими процессорами с целью повышения производительности. Эта методология уже давно применяется в научных расчетах для сокращения их времени. Новые достижения в разработке параллельных компьютеров общего назначения и методов параллельного программирования дали возможность привнести параллелизм в область работы с данными.

В параллельных системах баз данных управление базами данных сочетается с параллельной обработкой с целью повышения производительности и доступности. Заметим, что производительность была целью *машин баз данных* в 1980-х годах. Перед традиционным управлением базами данных уже давно стояла проблема «бутылочного горлышка ввода-вывода», возникающая из-за того, что время доступа к диску очень велико по сравнению со временем доступа к памяти (обычно в сотни тысяч раз больше). Первоначально конструкторы машин баз данных пытались решить эту проблему за счет специализированного оборудования, например включения устройств фильтрации данных в считывающие головки дисков. Но этот подход потер-

пел неудачу, потому что соотношение стоимость–производительность было хуже, чем у программных решений, которым проще было воспользоваться прогрессом полупроводниковых технологий. Идея перенести функции базы данных ближе к диску возродилась после включения универсальных микропроцессоров в дисковые контроллеры, что привело к появлению интеллектуальных дисков. Например, базовые функции, которые требуют дорогостоящего последовательного просмотра, например операции выборки с нечеткими предикатами, можно эффективно выполнить на уровне диска, поскольку это позволяет не перегружать память СУБД ненужными дисковыми блоками. Однако для использования интеллектуальных дисков СУБД необходимо модифицировать, в частности процессор запросов должен решать, нужно ли использовать функции диска. Поскольку стандартной технологии интеллектуальных дисков не существует, адаптация к разным технологиям негативно сказывается на переносимости СУБД.

Но важным результатом является общее решение проблемы бутылочного горлышка ввода-вывода. Смысл его в том, чтобы *увеличить пропускную способность ввода-вывода с помощью параллелизма*. Например, если база данных размера D хранится на диске с пропускной способностью T, то пропускная способность всей системы ограничена T. С другой стороны, если распределить базу данных по n дискам, каждый емкостью D/n и с пропускной способностью T' (желательно эквивалентной T), то в идеальном случае мы получим пропускную способность $n * T'$, которой с успехом может воспользоваться несколько процессоров (в идеале n). Отметим, что хранение базы данных в основной памяти – скорее дополняющее, чем альтернативное решение. В частности, проблемы «бутылочного горлышка доступа к памяти» в таких системах также можно решить с помощью распараллеливания. Поэтому конструкторы параллельных систем баз данных направили усилия на разработку программных решений для эксплуатации параллельных компьютеров.

Параллельную систему баз данных можно неформально определить как СУБД, реализованную на параллельном компьютере. Под это определение подпадают многочисленные варианты, начиная с прямолинейного переноса существующих СУБД, для чего, возможно, будет достаточно переписать только подпрограммы взаимодействия с операционной системой, до хитроумного сочетания параллельной обработки и функций системы баз данных в новой программно-аппаратной архитектуре. Как всегда, необходимо найти компромисс между переносимостью (на несколько платформ) и эффективностью. Хитроумный подход позволяет полнее задействовать возможности, предлагаемые мультипроцессором, но за счет переносимости. Интересно, что это дает различные преимущества производителям компьютеров и программного обеспечения. Поэтому важно охарактеризовать основные вехи в пространстве альтернативных архитектур параллельных систем. Для этого уточним, что представляет собой параллельная система баз данных и какими функциями она должна обладать. Это будет полезно для сравнения различных архитектур.

Цели параллельной и распределенной систем баз данных похожи (производительность, доступность, расширяемость), но из-за более тесной связи вычислительных и запоминающих узлов акценты несколько смещаются.

1. **Высокая производительность.** Ее можно достичь применением нескольких дополняющих друг друга решений: параллельное управление данными, оптимизация запросов и балансировка нагрузки. Распараллеливание может увеличить пропускную способность и уменьшить время ответа при обработке транзакций. Однако уменьшение времени ответа на сложный запрос посредством массового параллелизма может привести к увеличению полного времени обработки (за счет дополнительного взаимодействия) и в качестве побочного эффекта снизить пропускную способность. Поэтому так важно оптимизировать распараллеливание запросов, чтобы свести к минимуму накладные расходы, например ограничить степень параллелизма. Под *балансировкой нагрузки* понимается способность системы разделять данную рабочую нагрузку между всеми процессорами. В зависимости от архитектуры параллельной системы это можно делать статически, благодаря выбору физической структуры базы данных, или динамически во время выполнения.
2. **Высокая доступность.** В параллельной системе баз данных много избыточных компонентов, что способствует повышению доступности данных и отказоустойчивости. В системе с высокой степенью параллелизма, содержащей много узлов, вероятность отказа узла довольно высока. Реплицирование данных в нескольких узлах полезно для *отработки отказа* – так называется метод повышения отказоустойчивости, состоящий в автоматическом перенаправлении транзакций с отказавшего узла на другой, где хранится копия данных. В результате обслуживание пользователей не прерывается.
3. **Расширяемость.** В параллельной системе должно быть проще приспособиться к росту размера базы данных или повышению требований к производительности (пропускной способности). Расширяемость – это способность системы к плавному расширению путем добавления новых средств обработки и хранения. В идеале параллельная система баз данных должна обладать двумя характеристиками: *линейным ускорением* и *линейной вертикальной масштабируемостью* (см. рис. 8.1). Линейное ускорение означает, что производительность системы линейно возрастает при сохранении размера и нагрузки на базу данных и увеличении числа узлов (т. е. средств обработки и хранения). Линейная вертикальная масштабируемость означает, что производительность не уменьшается при одновременном линейном увеличении размера базы данных, нагрузки и числа узлов. Кроме того, реорганизация существующей базы данных при расширении системы должна быть минимальна.

Все более широкое распространение кластеров в крупномасштабных приложениях, например для управления веб-данными, привело к появлению терминов *горизонтальное масштабирование* и *вертикальное масштабирование*. На рис. 8.2 показан кластер с 4 серверами, каждый из которых имеет несколько процессорных узлов (Р). В этом контексте вертикальное масштабирование означает добавление новых узлов в сервер и ограничение максимальным размером сервера. А горизонтальное масштабирование означает добавление новых серверов, слабо связанных друг с другом, поэтому его возможности почти не ограничены.

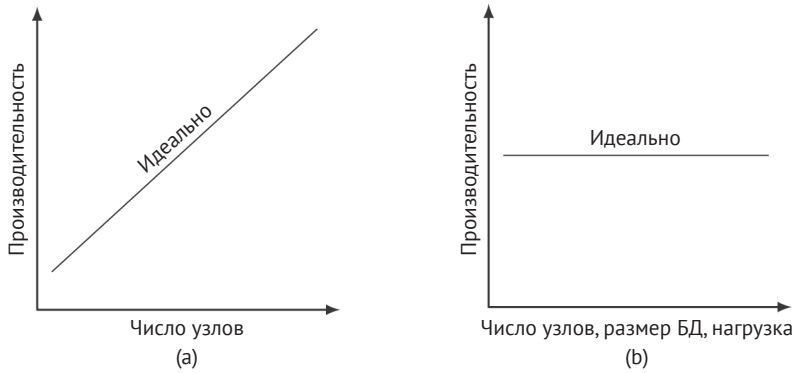


Рис. 8.1 ❖ Метрики расширяемости:
(a) линейное ускорение; (b) линейная вертикальная масштабируемость

Рис. 8.2 ❖ Сравнение вертикального
и горизонтального масштабирования

8.2. Параллельные архитектуры

При проектировании параллельной системы баз данных необходимо идти на компромиссы, чтобы обеспечить вышеупомянутые преимущества при хорошем соотношении стоимость–производительность. Одно из важнейших проектных решений – способ соединения основных элементов оборудования – процессоров, основной памяти и дисков – с помощью сети. В этом разделе мы опишем архитектурные аспекты параллельных систем баз данных, в частности рассмотрим и сравним три основные архитектуры: *с общей памятью*, *с общим диском* и *без разделения ресурсов*. Общая память применяется в сильно связанных мультипроцессорах, а архитектуры с общим диском и без разделения ресурсов – в кластерах. При описании этих архитектур нас будут интересовать четыре элемента оборудования: межсоединительная сеть, процессоры (P), модули основной памяти (M) и диски. Для простоты прочие

элементы, например процессорные кеши, процессорные ядра и шина ввода-вывода, игнорируются.

8.2.1. Общая архитектура

В предположении, что используется архитектура клиент-сервер, функции, поддерживаемые параллельной системой баз данных, можно распределить между тремя подсистемами, как в типичной СУБД. Различия же связаны с реализацией этих функций, которые должны учитывать параллелизм, секционирование и репликацию данных и распределенные транзакции. В зависимости от архитектуры процессорный узел может поддерживать все эти подсистемы или их часть. На рис. 8.3 показана архитектура вместе с этими подсистемами, которая основана на архитектуре, изображенной на рис. 1.11, с добавлением менеджера клиентов.

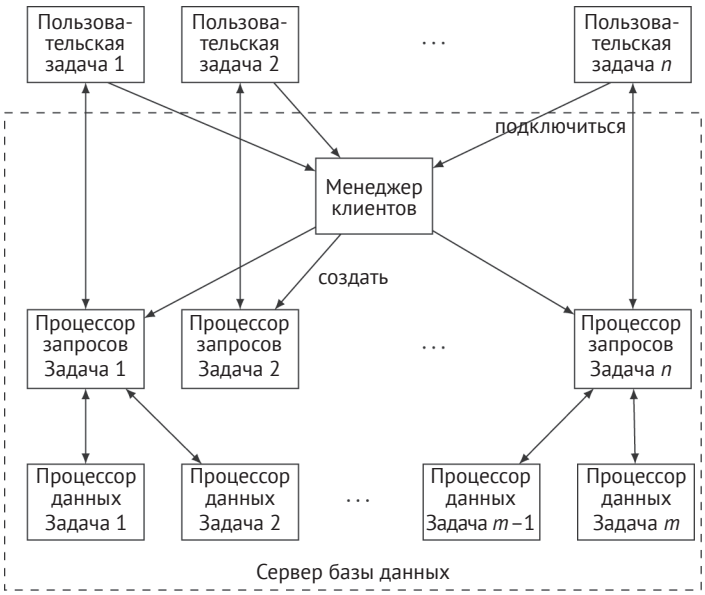


Рис. 8.3 ❖ Общая архитектура параллельной системы баз данных

1. **Менеджер клиентов** поддерживает взаимодействие клиентов с параллельной системой баз данных. В частности, он управляет подключениями и отключениями между клиентскими процессами, работающими на других серверах, например на серверах приложений, и процессорами запросов. Он инициирует клиентские запросы (которые могут быть транзакциями) на некоторых процессорах запросов, которые затем принимают на себя ответственность за прямое взаимодействие с клиентами и занимаются обработкой запросов и управлением транзакциями. Менеджер клиентов также выполняет балансировку нагрузки, пользуясь для этого каталогом, в котором хранится информация

о нагрузке на процессорные узлы и предкомпилированные запросы (включая местоположение данных). Это позволяет запускать предкомпилированный запрос на процессорах запросов, расположенных рядом с данными, к которым производится обращение. Менеджер клиентов – простой процесс, поэтому узким местом он не является. Однако для отказоустойчивости его можно реплицировать в нескольких узлах.

2. **Процессор запросов** получает от клиентов запросы и обрабатывает их: компилирует, выполняет, запускает транзакции. Хранит всю метаинформацию о данных, запросах и транзакциях в каталоге базы данных. Самим каталогом следует управлять как базой данных, т. е. реплицировать во всех узлах процессоров данных. В зависимости от запроса активирует различные этапы компиляции, в т. ч. семантический контроль данных, оптимизацию и распараллеливание, запускает и следит за выполнением запроса с помощью процессоров данных, возвращает результаты кодов ошибок клиентам. Может также запускать проверку транзакций на процессорах данных.
3. **Процессор данных** управляет данными в базе и системными данными (системным журналом и т. д.), а также предоставляет все низкоуровневые функции, необходимые для параллельного выполнения запросов: выполнение операторов базы данных, поддержку параллельных транзакций, управление кешем и т. д.

8.2.2. Архитектура с общей памятью

В архитектуре с общей памятью любой процессор имеет доступ к любому модулю памяти или диску по межсоединительной сети. Все процессоры управляются одной операционной системой.

Существенное преимущество такой архитектуры – простота модели программирования, основанной на общей виртуальной памяти. Поскольку метаинформация (каталог) и управляющая информация (например, таблицы блокировок) могут разделяться всеми процессорами, программное обеспечение базы данных пишется в основных чертах так же, как для компьютеров с одним процессором. В частности, межзапросный параллелизм достается даром. Для организации внутризапросного параллелизма требуются некоторые усилия, но не слишком большие. Балансировка нагрузки также осуществляется легко, поскольку это можно делать во время выполнения, распределяя новую задачу в общей памяти наименее занятому процессору.

В зависимости от того, как разделяется физическая память, выделяют два подхода: равномерный доступ к памяти (Uniform Memory Access – UMA) и неравномерный доступ к памяти (Non-Uniform Memory Access – NUMA).

8.2.2.1. Равномерный доступ к памяти (UMA)

В случае UMA физическая память разделяется всеми процессорами, так что время доступа к памяти постоянно (рис. 8.4). Поэтому такую архитектуру называют еще *симметричным мультипроцессором* (*SMP*). Для межсоедине-

ния процессоров применяются такие топологии, как шина, координатная коммутация и многосвязная сеть (mesh).

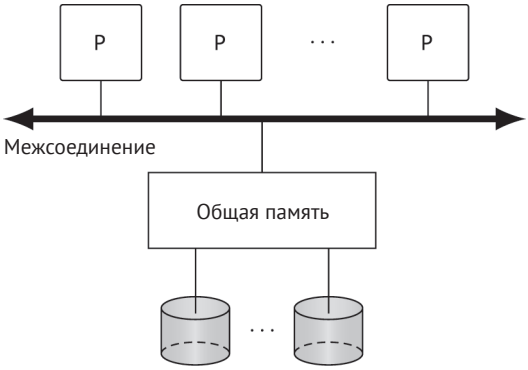


Рис. 8.4 ❖ Общая память

Первые SMP появились в 1960-х годах для мейнфреймов и содержали немного процессоров. В 1980-х годах были сконструированы SMP с десятками процессоров. Но они отличались высокой ценой и ограниченной масштабируемостью. Высокая цена была обусловлена сложностью межсоединений, поскольку каждый процессор нужно было соединить с каждым модулем памяти или диском. По мере того как процессоры становились все быстрее, даже при наличии больших кешей резко возрастало количество конфликтов доступа к памяти, что снижало производительность. Поэтому такие системы масштабировались не более чем на десяток процессоров. Наконец, поскольку пространство памяти разделялось всеми процессорами, сбой памяти мог отразиться на большинстве процессоров, что снижало доступность данных.

Многоядерные процессоры тоже основаны на SMP, но несколько процессорных ядер и общая память находятся на одном кристалле. По сравнению с предшествующими конструкциями с несколькими кристаллами, они повышают производительность операций кеширования, требуют меньше места на печатной плате и потребляют меньше энергии. Поэтому в настоящее время наблюдается тенденция к увеличению числа ядер, так что доступны уже процессоры с сотнями ядер.

Примерами параллельных систем баз данных с архитектурой SMP служат XPRS, DBS3 и Volcano.

8.2.2.2. Неравномерный доступ к памяти (NUMA)

Цель архитектуры NUMA – предоставить модель программирования с общей памятью со всеми ее достоинствами, но при этом обеспечить масштабируемую архитектуру с распределенной памятью. У каждого процессора имеется свой локальный модуль памяти, к которому он может обращаться очень эффективно. Термин NUMA отражает тот факт, что стоимость доступа к (виртуально) общей памяти различается в зависимости от того, произво-

дится обращение к локальной памяти процессора или удаленно к памяти другого процессора.

Самый старый класс NUMA-систем – мультипроцессоры NUMA с поддержкой когерентности кешей (Cache Coherent NUMA – CC-NUMA) (рис. 8.5). Поскольку различные процессоры могут обращаться к одним и тем же данным в режиме конфликтующих обновлений, необходимы протоколы глобальной согласованности кешей. Чтобы доступ к удаленной памяти был эффективен, согласованность кешей обеспечивается аппаратно с помощью специального межсоединения кешей. Поскольку общая память и согласованность кешей поддерживаются аппаратно, доступ к удаленной памяти очень эффективен, всего в несколько раз (обычно не более чем в три) медленнее локального.

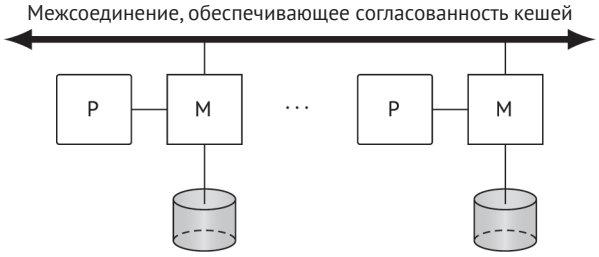


Рис. 8.5 ❖ Архитектура неравномерного доступа к памяти с поддержанием когерентности кешей (CC-NUMA)

Сравнительно недавний подход к NUMA основан на использовании технологии удаленного прямого доступа к памяти (Remote Direct Memory Access – RDMA), которая теперь поддерживается кластерными сетями межсоединения с низкой задержкой типа Infiniband и Myrinet. RDMA реализована на сетевой карте и обеспечивает передачу данных с нулевым копированием, т. е. узел кластера может напрямую обращаться к памяти другого узла без копирования в буферы операционной системы. В результате задержка типичной операции доступа к удаленной памяти превышает задержку локального доступа не более чем в 10 раз. Но возможности для совершенствования еще не исчерпаны. Например, более тесная интеграция управления удаленной памятью в иерархию локальной когерентности узла позволяет добиться задержки удаленного доступа, превышающей локальную не более чем в 4 раза. Таким образом, RDMA можно использовать для улучшения производительности операций в параллельной базе данных. Однако требуются новые алгоритмы с поддержкой NUMA, чтобы устранить узкое место в виде доступа к удаленной памяти. Базовый подход состоит в том, чтобы как можно чаще обращаться к локальной памяти, для чего выполнение задач СУБД планируется рядом с данными, а вычисления и обмен данными по сети чередуются.

В современных мультипроцессорах применяется иерархическая архитектура, в которой сочетаются NUMA и UMA, т. е. имеется мультипроцессор NUMA, в котором все процессоры многоядерные. Каждый такой мультипроцессор NUMA может играть роль узла кластера.

8.2.3. Архитектура с общим диском

В кластере с общим диском (рис. 8.6) у каждого процессора имеется доступ к любому диску по шине межсоединения, но только монопольный (ни с кем не разделяемый) доступ к своей основной памяти. Каждый узел с процессором и памятью управляется своим экземпляром операционной системы. Таким образом, каждый процессор может обращаться к страницам базы данных на общем диске и кешировать их в своей памяти. Поскольку разные процессоры могут обращаться к одной и той же странице в режиме конфликтующего доступа, необходимо поддерживать глобальную согласованность кешей. Обычно для этого используется распределенный диспетчер блокировок, который можно реализовать, применяя описанные в главе 5 методы. Первой параллельной СУБД с общим диском стал Oracle, в который была включена эффективная реализация распределенного диспетчера блокировок для обеспечения согласованности кешей. Эта реализация эволюционировала в движок базы данных Oracle Exadata. Другие крупные поставщики СУБД – IBM, Microsoft и Sybase – также предлагают реализации с общим диском, обычно для рабочих нагрузок типа OLTP.

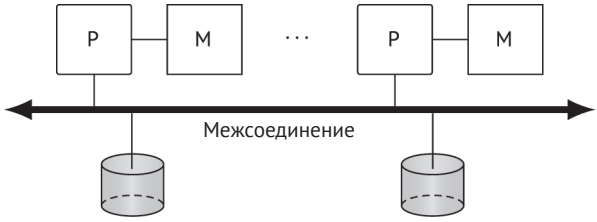


Рис. 8.6 ❖ Архитектура с общим диском

В архитектуре с общим диском требуется, чтобы диски были доступны всем узлам кластера. Существует две основные технологии разделения дисков в кластере: сетевое хранилище данных (network-attached storage – NAS) и сеть хранения данных (storage-area network – SAN). NAS – это специальное устройство с дисками, подключаемое к сети (обычно типа TCP/IP), доступ к которому производится по протоколу распределенной файловой системы типа NFS. NAS хорошо подходит для приложений, которым достаточно низкой пропускной способности, например резервного копирования и архивирования данных с жестких дисков ПК. Однако эта технология сравнительно медленная и непригодна для управления базой данных, т. к. если узлов много, то она быстро становится узким местом. Сеть хранения данных (SAN) предлагает похожую функциональность, но на более низком уровне. Для повышения эффективности в ней используется блочный протокол, поэтому поддерживать согласованность кешей (на уровне блоков) проще. В результате SAN обеспечивает высокую пропускную способность данных и может масштабироваться на большое число узлов.

Архитектура с общим диском имеет три основных преимущества: простое и дешевое администрирование, высокая доступность и хороший баланс на-

грузки. Администраторам базы данных не приходится иметь дело со сложным секционированием данных, а отказ одного узла влияет только на кешированные в нем данные, тогда как данные на диске остаются доступными всем остальным узлам. Балансировка нагрузки упрощается, поскольку любой запрос может быть обработан любым узлом с процессором и памятью. Основные недостатки – цена (из-за стоимости SAN) и ограниченная масштабируемость на очень большие базы данных вследствие накладных расходов на протоколы когерентности кешей. Решение – прибегнуть к секционированию данных, как в архитектуре без разделения ресурсов, правда, ценой более сложного администрирования.

8.2.4. Архитектура без разделения ресурсов

В кластере без разделения ресурсов (рис. 8.7) у процессора имеется монопольный доступ к своей основной памяти и диску, непосредственно подключенному к серверу (Directly Attached Storage – DAS).

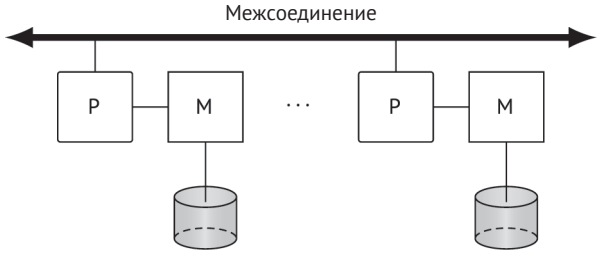


Рис. 8.7 ❖ Архитектура без разделения ресурсов

Каждый узел с процессором, памятью и диском управляется своим экземпляром операционной системы. Кластеры без разделения ресурсов широко применяются на практике, обычно с использованием NUMA-узлов, потому что обладают наилучшим соотношением стоимость–производительность и способны масштабироваться в очень широких пределах (тысячи узлов).

Каждый узел можно рассматривать как локальный узел (с собственной базой данных и программным обеспечением) распределенной СУБД. Поэтому применимо большинство решений, разработанных для таких систем, в т. ч. фрагментация базы данных, управление распределенными транзакциями и обработка распределенных запросов. Быстрая межсоединительная сеть позволяет подключать большое число узлов. В отличие от SMP, такую архитектуру часто называют массово параллельным процессором (Massively Parallel Processor – MPP).

Системы без разделения ресурсов допускают плавный постепенный рост путем добавления новых узлов и таким образом обеспечивают расширяемость и масштабируемость. Однако требуется тщательно продумывать секционирование данных на нескольких дисках. Кроме того, после добавления новых узлов, вероятно, понадобится заново секционировать базу данных,

чтобы решить проблемы балансировки нагрузки. Наконец, обеспечить отказоустойчивость трудно (необходима репликация), поскольку после отказа узла данные на его диске становятся недоступными.

Архитектура без разделения ресурсов принята во многих прототипах параллельных систем баз данных, например: Bubba, Gamma, Grace и Prisma/DB. Первой серьезной параллельной СУБД стала машина баз данных Teradata. Другие крупные поставщики СУБД, включая IBM, Microsoft, Sybase, а также поставщики столбцовых СУБД типа MonetDB и Vertica предлагают реализации без разделения ресурсов для OLAP-приложений из верхнего ценового сегмента. Наконец, в СУБД типа NoSQL и в системах больших данных, как правило, используется архитектура без разделения ресурсов.

Отметим, что возможна также гибридная архитектура, когда часть кластера работает без разделения ресурсов, например для рабочих нагрузок типа OLAP, а часть имеет общий диск, например для рабочих нагрузок типа OLTP. Так, Teradata поддерживает концепцию *клики*, т. е. набора узлов с общим набором дисков в дополнение к своей архитектуре без разделения ресурсов; цель – повысить доступность.

8.3. Размещение данных

В оставшейся части этой главы мы будем рассматривать архитектуру без разделения ресурсов, поскольку это самый общий случай, а разработанные для нее методы реализации применимы, иногда в более простой форме, к другим архитектурам. Размещение данных в параллельной системе баз данных во многом схоже с фрагментацией данных в распределенных базах (см. главу 2). Очевидное сходство состоит в том, что фрагментацию можно использовать для повышения степени параллелизма. Как было отмечено в главе 2, в параллельных СУБД чаще всего применяется горизонтальное секционирование, хотя может использоваться и вертикальное – для повышения степени параллелизма и балансировки нагрузки, как в распределенных базах данных и в столбцовых СУБД типа MonetDB или Vertica. Есть и еще одно сходство с распределенными базами данных – поскольку данные гораздо больше по объему, чем программы, выполнение должно по возможности происходить там, где находятся данные. В главе 2 мы отмечали два важных различия между параллельными и распределенными базами данных. Во-первых, нет необходимости максимизировать локальную обработку (в каждом узле), потому что между пользователями и узлами нет никакой связи. Во-вторых, добиться балансировки нагрузки при большом количестве узлов гораздо труднее. Основная проблема в том, чтобы избежать состязания за ресурсы, которое могло бы привести к пробуксовке системы в целом (например, один узел делает всю работу, а остальные простаивают). Поскольку программы выполняются там, где находятся данные, размещение данных играет критически важную роль с точки зрения производительности.

В параллельных СУБД наиболее распространенными стратегиями секционирования данных являются циклическая, хешированием и по диапазонам

(см. раздел 2.1.1). Секционирование данных должно масштабироваться с увеличением размера базы данных и нагрузки на нее. Таким образом, степень секционирования, т. е. количество узлов, в которых хранится отношение, должна быть функцией от размера и частоты доступа к отношению. Следовательно, увеличение степени секционирования может повлечь за собой реорганизацию размещения. Например, если мощность отношения, которое первоначально было размещено в восьми узлах, удвоилась в результате вставок, то теперь оно должно быть размещено в 16 узлах.

В системах с высокой степенью параллелизма и секционированием данных для балансировки нагрузки необходима периодическая реорганизация, которую нужно проводить часто, если только рабочая нагрузка не является статической и обновлений немного. Реорганизации должны быть прозрачны для откомпилированных запросов, выполняемых на сервере базы данных. В частности, реорганизация не должна быть причиной перекомпиляции запросов, которые должны оставаться независимыми от часто меняющегося местоположения данных. Такой независимости можно достичь, если исполняющая система поддерживает ассоциативный доступ к распределенным данным. Это отличается от распределенной СУБД, где ассоциативный доступ реализуется во время компиляции процессором запросов, который пользуется каталогом данных.

Один из возможных способов организации ассоциированного доступа – завести глобальный индекс, реплицируемый в каждом узле. Глобальный индекс показывает размещение отношения на множестве узлов. Концептуально это двухуровневый индекс, с главной кластеризацией по имени отношения и дополнительной по какому-то атрибуту отношения. Этот глобальный индекс поддерживает переменное секционирование, когда степени секционирования отношений могут различаться. Индекс может быть организован как хеш-таблица или как B-дерево. В обоих случаях запросы с точным совпадением можно эффективно обработать в одном узле. Но в случае хеш-таблицы для обработки запросов по диапазону необходимо просматривать все узлы, где хранятся указанные в запросе данные. B-дерево (которое обычно занимает гораздо больше места, чем хеш-индекс) позволяет обрабатывать запросы по диапазону более эффективно, поскольку просматриваются только узлы, содержащие данные из указанного диапазона.

Пример 8.1. На рис. 8.8 приведен пример глобального и локального индексов для отношения `EMP(ENO, ENAME, TITLE)` из базы данных конструкторской компании.

Пусть требуется найти элементы отношения `EMP`, для которых атрибут `ENO` равен «E50». Индекс первого уровня отображает имя `EMP` на индекс по `ENO` над `EMP`. Затем индекс второго уровня отображает кластерное значение «E50» на узел с номером *j*. Необходим также локальный индекс в каждом узле, который отображает отношение на множество дисковых страниц в этом узле. Локальный индекс двухуровневый, с главной кластеризацией по имени отношения и дополнительной по какому-то атрибуту. Дополнительный кластерный атрибут в локальном индексе *такой же*, как в глобальном. Таким образом, *ассоциативная маршрутизация* уточняется при переходе от одного

узла к другому по паре (имя отношения, кластерное значение). Этот локальный индекс отображает кластерное значение «E5» на страницу 91. ♦

Рис. 8.8 ❖ Пример глобального и локального индексов

Серьезная проблема при размещении данных возникает в случае асимметричных распределений данных, что может стать причиной неравномерного секционирования и негативно сказаться на сбалансированности нагрузки. Решение состоит в том, чтобы обращаться с неравномерными секциями специальным образом, например путем дальнейшего дробления больших секций. Это легко сделать в случае секционирования по диапазонам, т. к. секцию можно разделить, как лист B-дерева, проведя реорганизацию локального индекса. В случае хеширования нужно использовать другую хеш-функцию по иному атрибуту. Здесь полезно провести различие между логическими и физическими узлами, допустив, что одному логическому узлу может соответствовать несколько физических.

И последний фактор, осложняющий размещение данных, – репликация для обеспечения высокой доступности, которую мы подробно обсуждали в главе 6. В параллельной СУБД возможны упрощенные подходы, например архитектура с *зеркалированными дисками*, в которой поддерживаются две копии данных: главная и резервная. Но в случае отказа узла нагрузка на узел, содержащий копию, может удвоиться, ухудшив сбалансированность нагрузки. Чтобы избежать этой проблемы, для параллельных систем баз данных было предложено несколько стратегий репликации, обеспечивающих высокую доступность. Интересное решение реализовано в Teradata: чередующееся секционирование, когда резервная копия сама секционируется на нескольких узлах. На рис. 8.9 показано чередующееся секционирование отношения R на четыре узла, в котором каждая главная копия секции, например R_1, дробится на три секции, например $R_{1,1}$, $R_{1,2}$ и $R_{1,3}$, и все они располагаются в разных резервных узлах. В случае отказа нагрузка на главную копию балансируется между узлами резервной копии. Но если откажут два узла, то доступ к отношению теряется, т. е. страдает доступность. Реконструкция главной копии из разделенных резервных копий может стоить дорого. В нор-

мальном режиме поддержание согласованности копии тоже может оказаться недешевой операцией.

Узел	1	2	3	4
Главная копия	R_1	R_2	R_3	R_4
Резервные копии	$R_{2,1}$ $R_{3,1}$ $R_{4,1}$	$R_{1,1}$ $R_{3,2}$ $R_{4,2}$	$R_{1,2}$ $R_{2,2}$ $R_{4,3}$	$R_{1,3}$ $R_{2,3}$ $R_{3,3}$

Рис. 8.9 ❖ Пример чередующегося секционирования

Альтернативное решение – *цепное секционирование* в системе Gamma, когда главная и резервная копии хранятся в двух соседних узлах (рис. 8.10). Идея в том, что вероятность выхода из строя двух соседних узлов гораздо ниже, чем вероятность выхода из строя двух произвольных узлов. В случае отказа нагрузка на отказавший и резервный узлы балансируется между всеми оставшимися узлами благодаря использованию узлов главной и резервной копий. Кроме того, поддержание согласованности копий обходится дешевле. Остается открытым вопросом, как выполнить размещение данных, принимая во внимание репликацию. По аналогии с размещением фрагментов в распределенной базе данных, эту задачу следует рассматривать как задачу оптимизации.

Узел	1	2	3	4
Главная копия	R_1	R_2	R_3	R_4
Резервные копии	R_4	R_1	R_2	R_3

Рис. 8.10 ❖ Пример цепного секционирования

8.4. Параллельная обработка запросов

Цель параллельной обработки запроса – преобразовать его в план выполнения, который можно эффективно выполнить параллельно. Для этого используется параллельное размещение данных и различные формы параллелизма, предлагаемые высокоуровневыми запросами. В этом разделе мы сначала познакомимся с базовыми параллельными алгоритмами обработки данных, а затем обсудим оптимизацию параллельной обработки запросов.

8.4.1. Параллельные алгоритмы обработки данных

Секционированное размещение данных – основа параллельного выполнения запросов к базе данных. Если дано секционированное размещение, то важно спро-

ектировать эффективные параллельные алгоритмы для обработки операторов базы данных (т. е. операторов реляционной алгебры) и запросов, включающих несколько операторов. Это трудная задача, т. к. требуется найти компромисс между параллелизмом и затратами на передачу данных между процессорами, которые могут быть тем выше, чем больше степень параллелизма.

Параллельные алгоритмы для операторов реляционной алгебры – основа параллельной обработки запросов. Их цель – максимизировать степень параллелизма. Однако, согласно закону Амдала, распараллелить можно лишь часть алгоритма. Обозначим *seq* относительную долю последовательной части программы (это число от 0 до 1), которая не может быть распараллелена, и пусть p – число процессоров. Тогда максимальное ускорение, достигаемое в результате распараллеливания, равно

$$MaxSpeedup(seq, p) = \frac{1}{seq + \left(\frac{1 - seq}{p}\right)}.$$

Например, если $seq = 0$ (вся программа параллельна) и $p = 4$, то получается идеальное ускорение 4. Но если $seq = 0.3$, то ускорение составляет всего 2.1. И даже если удвоить количество процессоров, т. е. положить $p = 8$, то ускорение возрастет совсем чуть-чуть – до 2.5. Поэтому при проектировании параллельных алгоритмов обработки данных важно свести к минимуму последовательную часть алгоритма и максимизировать параллельную, задействовав внутриоператорный параллелизм.

Обработка оператора выборки в контексте секционированного размещения данных производится так же, как во фрагментированной распределенной базе. В зависимости от предиката выборки оператор может быть выполнен в одном узле (если это предикат точного равенства) или во всех узлах, на которые секционировано отношение, если предикат достаточно сложный. Если глобальный индекс организован в виде структуры, напоминающей B-дерево (рис. 8.8), то выполнение оператора выборки с предикатом по диапазону можно ограничить только узлами, где хранятся релевантные данные. Далее в этом разделе мы займемся параллельной обработкой двух важнейших операторов, встречающихся в запросах: сортировки и соединения.

8.4.1.1. Параллельные алгоритмы сортировки

Сортировка отношений необходима в запросах, где требуется получить упорядоченный результат, а также для агрегирования и группировки. Выполнить ее эффективно трудно, потому что каждый элемент нужно сравнить с каждым. Один из самых быстрых алгоритмов сортировки на одном процессоре, *quicksort*, в значительной своей части последовательный, поэтому, согласно закону Амдала, не годится для распараллеливания. Один из самых популярных параллельных алгоритмов – параллельная сортировка слиянием, т. к. его легко реализовать и он не предъявляет особых требований к архитектуре параллельной системы. Поэтому он используется в кластерах с общим диском и без разделения ресурсов. Также его можно адаптировать для работы на многоядерных процессорах.

Кратко опишем b-путевой алгоритм сортировки слиянием. Пусть сортируемое множество состоит из n элементов. Определим отрезок как упорядоченную последовательность элементов; таким образом, наше множество содержит n одноэлементных отрезков. Алгоритм заключается в итеративном слиянии b отрезков из K элементов в отсортированный отрезок из $K * b$ элементов, начиная с $K = 1$. На i-м проходе каждый набор b отрезков из b_{i-1} элементов сливается в отсортированный отрезок из b_i элементов. Если начать с $i = 1$, то число проходов, необходимых для сортировки n элементов, равно $\log_b n$.

Теперь опишем применение этого метода в кластере без разделения ресурсов. Предположим, что для выполнения параллельных задач используется популярная модель главный–подчиненный, когда один главный узел координирует работу подчиненных узлов, отправляя им задачи и данные и получая в ответ уведомления о том, что задача выполнена.

Пусть требуется отсортировать отношение, занимающее p дисковых страниц и секционированное на n узлов. У каждого узла имеется локальная память из $b + 1$ страниц, где b страниц используется для входа, а одна – для выхода. Алгоритм состоит из двух этапов. На первом этапе каждый узел локально сортирует свой фрагмент, например используя алгоритм quicksort, если узел оснащен одним процессором, или алгоритм b-путевой сортировки слиянием, если в узле установлен многоядерный процессор. Этот этап называется оптимальным, потому что все узлы полностью загружены. В результате генерируется n отрезков по p/n страниц, и если n равно b, то один узел может слить их за один проход. Однако n может быть гораздо больше b, и тогда главный узел должен организовать множество рабочих узлов в виде дерева порядка b на последнем этапе, называемом постоптимальным. Количество необходимых узлов уменьшается в b раз на каждом проходе. На последнем проходе один узел сливает все отношение. Количество проходов на постоптимальном этапе равно $\log_b p$. Из-за этого этапа степень параллелизма уменьшается.

8.4.1.2. Параллельные алгоритмы соединения

Для соединения двух произвольных секционированных отношений можно воспользоваться одним из трех базовых параллельных алгоритмов: соединение с помощью параллельной сортировки слиянием, параллельные вложенные циклы (parallel nested loop – PNL) и параллельное хеш-соединение (parallel hash join – PHJ). Это варианты соответствующих централизованных аналогов. Алгоритм соединения с помощью параллельной сортировки слиянием просто сортирует оба отношения по атрибуту соединения, применяя параллельную сортировку слиянием, и соединяет их на одном узле, пользуясь операцией, напоминающей слияние. Последняя операция последовательная, но зато отношение, являющееся результатом соединения, отсортировано по атрибуту соединения, что может оказаться полезным для следующей операции.

Два других алгоритма полностью параллельны. Мы опишем их на псевдоконкурентном языке программирования, в котором имеется три основные конструкции: `parallel-do`, `send` и `receive`. `Parallel-do` говорит, что следующий блок действий выполняется параллельно. Например, предложение

```
for i from 1 to n in parallel-do action A
```

означает, что действие A следует выполнить параллельно n узлами. Конструкции send и receive – базовые примитивы передачи данных: send отправляет данные одному или нескольким узлам, а receive получает данные, отправленные конкретным узлом. Далее мы будем рассматривать соединение двух отношений R и S, секционированных на m и n узлов соответственно. Для простоты предположим, что эти m и n узлов не пересекаются. Узел, в котором размещается фрагмент отношения R (соответственно S), называется R-узлом (соответственно S-узлом).

Параллельный алгоритм вложенных циклов

Параллельный алгоритм вложенных циклов простой и общий. Он реализует метод фрагментации и репликации, описанный в разделе 4.5.1. По существу, он параллельно строит декартово произведение отношений R и S. Поэтому поддерживаются сколько угодно сложные предикаты соединения, а не только соединение по равенству.

Алгоритм выполняет два вложенных цикла. Одно отношение выбирается в качестве внутреннего, к нему производится обращение во внутреннем цикле, а другое – в качестве внешнего и обрабатывается во внешнем цикле. Выбор зависит от функции стоимости с двумя основными группами параметров: размеры отношений, от которых зависят затраты на передачу данных, и наличие индексов по атрибутам соединения, от чего зависит стоимость локального соединения.

Алгоритм описан в алгоритме 8.1 для случая, когда результат формируется в S-узлах, т. е. S выбрано в качестве внутреннего отношения. Алгоритм состоит из двух этапов.

Алгоритм 8.1. Параллельные вложенные циклы (PNL)

Вход: $R_1, R_2, \ldots, R_m$: фрагменты отношения R;
$S_1, S_2, \ldots, S_n$: фрагменты отношения S;
JP: предикат соединения
Выход: $T_1, T_2, \ldots, T_n$: фрагменты результата
begin
  for *i от* 1 *до m параллельно* **do** {отправить R целиком на каждый S-узел}
    отправить R_i на каждый узел, содержащий фрагмент S
  end for
  for *j от* 1 *до n параллельно* **do** {выполнить соединение в каждом S-узле}
    $R \leftarrow \bigcup_{i=1}^{m} R_i$; {$R_i$ из R-узлов; R полностью реплицировано в S-узлах}
    $T_j \leftarrow R \bowtie_{JP} S_j$
  end for
end

На первом этапе каждый фрагмент R посылается всем узлам, содержащим фрагменты S (всего таких узлов n) и там реплицируется. Этот этап выполняется параллельно m узлами, т. е. необходимо отправить $m * n$ сообщений.

На втором этапе каждый S-узел j получает отношение R целиком и локально соединяет R с фрагментом S_j. Этот этап параллельно выполняется n узлами. Локальное соединение можно выполнить, как в централизованной

СУБД. В зависимости от алгоритма локального соединения обработка соединения может начаться сразу после получения данных или позже. Если используется алгоритм вложенных циклов, возможно с индексом по атрибуту соединения над S, то обработку соединения можно выполнять конвейерным способом сразу по поступлении кортежа R. С другой стороны, если используется алгоритм соединения с помощью сортировки слиянием, то все данные должны быть получены до того, как начнется соединение отсортированных отношений.

Короче говоря, параллельный алгоритм вложенных циклов можно рассматривать как замену оператора $R \bowtie S$ на $\bigcup_{i=1}^{n}(R \bowtie S_i)$.

Пример 8.2. На рис. 8.11 показано применение параллельного алгоритма вложенных циклов при $m = n = 2$. ♦

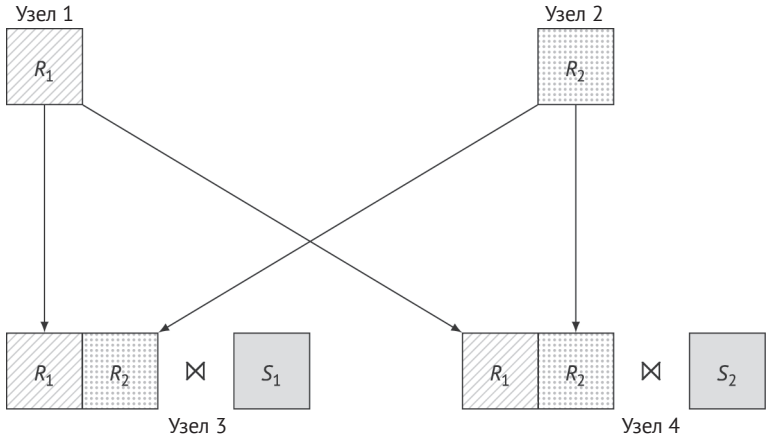


Рис. 8.11 ❖ Пример параллельных вложенных циклов

Параллельный алгоритм хеш-соединения

Параллельный алгоритм хеш-соединения, показанный в алгоритме 8.2, применим только в случае эквисоединения и не требует никакого особого секционирования отношений-операндов. Впервые он был предложен для машины баз данных Grace и известен как хеш-соединение Grace.

Идея заключается в том, чтобы секционировать R и S на одно и то же число p взаимно непересекающихся множеств (фрагментов) $R_1, R_2, \ldots, R_p$ и $S_1, S_2, \ldots, S_p$, так что

$$R \bowtie S = \bigcup_{i=1}^{p}(R_i \bowtie S_i).$$

Секционирование R и S основано на применении одной и той же хеш-функции к атрибуту соединения. Отдельные соединения $(R_i \bowtie S_i)$ производятся параллельно, и результат соединения порождается в p узлах. Эти p узлов могут быть даже выбраны во время выполнения в зависимости от нагрузки на систему.

Алгоритм 8.2. Параллельное хеш-соединение (PHJ)

Вход: $R_1, R_2, \ldots, R_m$: фрагменты отношения R;
$S_1, S_2, \ldots, S_n$: фрагменты отношения S;
JP: предикат соединения;
h: хеш-функция, возвращающая элемент [1, *p*]
Выход: $T_1, T_2, \ldots, T_p$: фрагменты результата
begin
  {Этап построения}
  for *i от* 1 *до m параллельно* **do**
    $R_i^j \leftarrow$ применить h(A) к R_i (j = 1, ..., p); {хешировать R на A)}
    отправить R_i^j узлу *j*
  end for
  for *j от* 1 *до p параллельно* **do**
    $R_j \leftarrow \bigcup_{i=1}^{m} R_j^i$ {получить фрагменты R_j от R-узлов}
    построить локальную хеш-таблицу для R_j
  end for
  {Этап апробирования}
  for *i от* 1 *до n параллельно* **do**
    $S_i^j \leftarrow$ применить h(B) к S_i (j = 1, ..., p); {хешировать S на B)}
    Отправить S_i^j узлу *j*
  end for
  for *j от* 1 *до p параллельно* **do**
    $S_j \leftarrow \bigcup_{i=1}^{n} S_j^i$ {получить фрагменты S_j от S-узлов}
    $T_j \leftarrow R_j \bowtie_{JP} S_j$ {апробировать S_j для каждого кортежа R_j }
  end for
end

Алгоритм состоит из двух этапов: *построения* и *апробирования*. На этапе построения отношение R, используемое как внутреннее, хешируется по атрибуту соединения и передается *p* целевым узлам, которые строят хеш-таблицу входящих кортежей. На этапе апробирования хешированное внешнее отношение S передается *p* целевым узлам, которые ищут (апробируют) каждый поступающий кортеж в хеш-таблице. Таким образом, как только хеш-таблицы для R и S построены, кортежи можно передавать и обрабатывать в конвейерном режиме, апробируя эти хеш-таблицы.

Пример 8.3. На рис. 8.12 показано применение параллельного алгоритма хеш-соединения при $m = n = 2$. Предполагается, что результат порождается в узлах 1 и 2. Поэтому стрелка из узла 1 в узел 1 или из узла 2 в узел 2 обозначает локальную передачу. ♦

Параллельный алгоритм хеш-соединения обычно гораздо эффективнее, чем параллельный алгоритм вложенных циклов, поскольку требует меньше передач данных и меньше работы для локального соединения на этапе апробирования. Кроме того, одно отношение, скажем R, может быть уже секционировано хешированием по атрибуту соединения. В таком случае этап построения вообще не нужен, и хешированные фрагменты S просто посылаются соответствующим R-узлам. В общем случае это более эффективно, чем параллельный алгоритм соединения с помощью сортировки слиянием.

Однако последний алгоритм все равно полезен, потому что порождает отношение, отсортированное по атрибуту соединения.

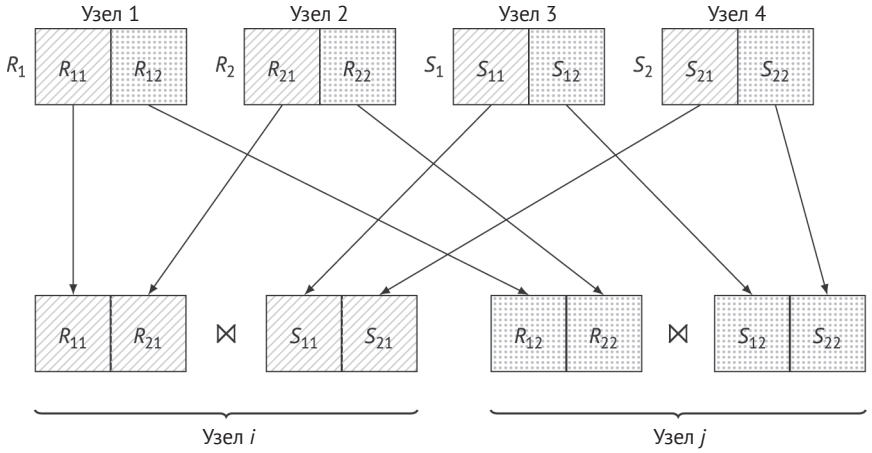


Рис. 8.12 ❖ Пример параллельного хеш-соединения

Проблема параллельного алгоритма хеш-соединения и его многочисленных вариантов заключается в том, что распределение значений атрибута соединения может быть асимметрично, а это ведет к несбалансированной нагрузке. Решения этой проблемы мы обсудим в разделе 8.5.2.

Варианты

Существует много вариантов базовых параллельных алгоритмов соединения – в частности, для адаптивной обработки запросов или созданных специально для баз данных в основной памяти либо многоядерных процессоров. Мы обсудим эти расширения ниже.

При рассмотрении адаптивной обработки запросов (см. раздел 4.6) проблема заключается в том, чтобы динамически упорядочить конвейерные операторы соединения во время выполнения, когда поступают кортежи из разных отношений. В идеале, когда приходит кортеж участвующего в соединении отношения, его следует отправить оператору соединения для обработки на лету. Но большинство алгоритмов соединения не могут обработать некоторые входящие кортежи на лету, потому что кортежи внутреннего и внешнего отношений обрабатываются асимметрично. Рассмотрим, к примеру, алгоритм PHJ: внутреннее отношение читается полностью на этапе построения, когда строится хеш-таблица, тогда как кортежи внешнего отношения можно подавать конвейером на этапе апробирования. Таким образом, входящий внутренний кортеж нельзя обработать на лету, т. к. он должен быть сохранен в хеш-таблице, и обработка станет возможна, только когда вся хеш-таблица будет построена. Точно так же несимметричен алгоритм соединения методом вложенных циклов, поскольку только внутреннее отношение должно быть прочитано целиком, а кортежи внешнего отношения можно обрабатывать конвейером. Алгоритмы соединения с той или иной асиммет-

рией оставляют мало возможностей для чередования ролей внутреннего и внешнего отношений. Следовательно, чтобы не так сильно зависеть от порядка подачи входных данных, нужны симметричные алгоритмы соединения, в которых изменение роли отношений в соединении не приводит к некорректным результатам.

Ранний пример симметричного алгоритма соединения дает симметричное хеш-соединение, в котором используются две хеш-таблицы, по одной для каждого входного отношения. Традиционные этапы построения и апробирования, присутствующие в базовом алгоритме хеш-соединения, просто чередуются. Поступающий кортеж одного отношения используется для апробирования хеш-таблицы, соответствующей другому отношению, и поиска соответствующих ему кортежей. Затем он вставляется в свою хеш-таблицу, так чтобы можно было соединить кортежи из другого отношения, которые поступят позже. Таким образом, любой поступающий кортеж можно обработать на лету. Популярен также пульсирующий алгоритм симметричного соединения (ripple join) – обобщение алгоритма вложенных циклов, в котором роли внутреннего и внешнего отношений постоянно меняются местами во время выполнения. Идея в том, чтобы сохранять состояния апробирования каждого входного отношения и указатель на последний кортеж, с помощью которого апробировалось другое отношение. В каждой точке пульсации внутреннее и внешнее отношения меняются ролями. В этот момент новое внешнее отношение приступает к апробированию входных кортежей внутреннего отношения, начиная с текущей позиции своего указателя, на протяжении заданного количества кортежей. В свою очередь, внутреннее отношение просматривается с первого кортежа до позиции его указателя минус 1. Количество кортежей внешнего отношения, обрабатываемых на каждом этапе, определяет частоту пульсации и может изменяться адаптивно.

Использование основной памяти процессоров также важно для повышения производительности параллельных алгоритмов соединения. Гибридный алгоритм хеш-соединения улучшает хеш-соединение Grace за счет использования доступной памяти для размещения всей секции (называемой секцией 0) в процессе секционирования, избегая тем самым обращений к диску. Еще один вариант – модифицировать этап построения, так чтобы результирующие хеш-таблицы помещались в основную память процессора. Это заметно улучшает производительность, поскольку уменьшается количество непопаданий в кеш на этапе апробирования хеш-таблицы. Та же идея используется в алгоритме поразрядного хеш-соединения для многоядерных процессоров, где доступ к памяти ядра производится гораздо быстрее, чем к удаленной общей памяти. Многопроходная схема секционирования применяется для разбиения обоих входных отношений на непересекающиеся секции по атрибуту соединения, так чтобы они помещались в память ядер. Затем над каждой секцией внутреннего отношения строятся хеш-таблицы, которые апробируются данными из соответствующей секции внешнего отношения. Параллельный алгоритм соединения с помощью сортировки слиянием, который, вообще говоря, считается менее эффективным, чем параллельное хеш-соединение, тоже можно оптимизировать для многоядерных процессоров.

8.4.2. Оптимизация параллельных запросов

Оптимизация параллельных запросов во многом схожа с распределенной обработкой запросов. Однако гораздо больше внимания уделяется извлечению преимуществ из внутриоператорного (благодаря применению описанных выше алгоритмов) и межоператорного параллелизма. Оптимизатор параллельных запросов, как и любой другой оптимизатор, состоит из трех компонентов: пространство поиска, модель стоимости и стратегия поиска. В этом разделе мы опишем эти компоненты в параллельном случае.

8.4.2.1. Пространство поиска

Для абстрагирования планов служат деревья операторов, определяющие порядок выполнения операторов. Деревья снабжены *аннотациями*, в которых указаны дополнительные аспекты выполнения, например алгоритм, реализуемый каждым оператором. В параллельных СУБД важным аспектом, который необходимо отразить в аннотациях, является тот факт, что два соседних оператора можно выполнить в *конвейерном* режиме. В этом случае второй оператор может начинаться раньше, чем закончится первый. Иными словами, второй оператор начинает *потреблять* кортежи, как только они *произведены* первым. В случае конвейерного выполнения не требуется материализовывать временные отношения, т. е. узел дерева, соответствующий конвейерно выполняемому оператору, не *хранится*.

Для некоторых операторов и алгоритмов один операнд необходимо сохранять. Например, на этапе построения в PHJ (алгоритм 8.2) хеш-таблица для меньшего отношения по атрибуту соединения строится параллельно. На этапе апробирования, когда большее отношение просматривается последовательно, для каждого его кортежа производится поиск в этой хеш-таблице. Поэтому конвейер и присоединенные к дереву аннотации ограничивают *диспетчеризацию* планов выполнения, разбивая дерево операторов на непересекающиеся поддеревья, соответствующие стадиям выполнения. Конвейерные операторы выполняются на одной и той же стадии, которая обычно называется *конвейерной цепочкой*, а операция сохранения определяет границы между двумя соседними стадиями.

Пример 8.4. На рис. 8.13 показано два дерева выполнения: без конвейера (рис. 8.13a) и с конвейером (рис. 8.13b). На рис. 8.13a необходимо полностью создать временное отношение `Temp1` и построить хеш-таблицу на этапе `Build2`, прежде чем на этапе `Probe2` можно будет начать потребление R_3. То же самое относится к `Temp2`, `Build3` и `Probe3`. Таким образом, имеем четыре стадии выполнения дерева: (1) построить хеш-таблицу R_1, (2) апробировать ее с помощью R_2 и построить хеш-таблицу `Temp1`, (3) апробировать ее с помощью `R3` и построить хеш-таблицу `Temp2`, (4) апробировать ее с помощью R_4 и произвести результат. На рис. 8.13b показано конвейерное выполнение. Если для построения хеш-таблиц хватает памяти, то выполнение дерева можно свести к двум стадиям: (1) построить таблицы для R_1, R_3 и R_4, (2) выполнить `Probe1`, `Probe2` и `Probe3` в конвейерном режиме. ♦

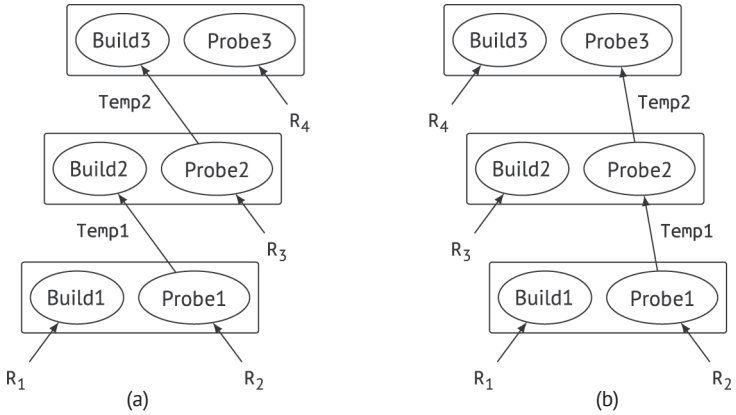


Рис. 8.13 ❖ Два дерева хеш-соединения с разной диспетчеризацией: (a) без конвейера; (b) конвейер, состоящий из R_2, Temp1 и Temp2

Множество узлов, в которых сохраняется отношение, называется его *домом*. *Дом оператора* – это множество узлов, где он выполняется, оно должно быть домом его операндов, чтобы оператор мог обратиться к операнду. Для бинарных операторов, например соединения, это условие иногда означает, что один из операндов нужно заново секционировать. Бывает даже, что оптимизатор считает выгодным заново секционировать оба операнда. В деревьях операторов присутствуют аннотации, указывающие на необходимость повторного секционирования.

На рис. 8.14 показано четыре дерева операторов, представляющих планы выполнения трехстороннего соединения. Деревья операторов могут быть *линейными*, когда по меньшей мере один операнд в каждом узле соединения является базовым отношением, или *кустистыми*. Удобно представлять конвейер отношений правой ветвью оператора. Тогда деревья, растущие вправо, соответствуют полностью конвейерному выполнению, а растущие влево – материализации всех промежуточных результатов. Таким образом, в предположении, что памяти достаточно для хранения отношений в левой части, длинные растущие вправо деревья более эффективны, чем соответствующие растущие влево деревья. В дереве, растущем влево, как на рис. 8.14a, только последний оператор может потреблять свое правое входное отношение в конвейерном режиме, при условии что левое входное отношение целиком помещается в оперативной памяти.

Интересны также формы параллельных деревьев, отличных от растущих влево и вправо. Например, только кустистые деревья (рис. 8.14d) допускают независимый параллелизм и отчасти конвейерный параллелизм. Независимый параллелизм полезен, когда отношения секционированы по непересекающимся домам. Предположим, что отношения на рис. 8.14d секционированы так, что у R_1 и R_2 общий дом h_1, а у R_3 и R_4 – общий дом h_2, отличный от h_1. Тогда оба соединения базовых отношений можно выполнить независимо и параллельно на множествах узлов, составляющих h_1 и h_2.

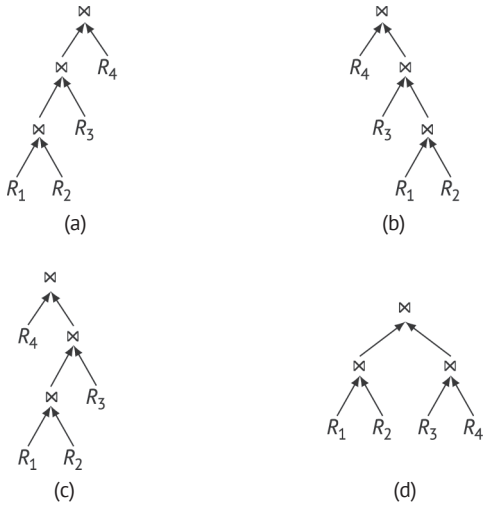


Рис. 8.14 ❖ Планы выполнения, представленные деревьями операторов: (a) растущее влево; (b) растущее вправо; (c) зигзагообразное; (d) кустистое

Если конвейерный параллелизм выгоден, то *зигзагообразные деревья*, занимающие промежуточное положение между растущими влево и вправо, иногда оказываются производительнее растущих вправо деревьев благодаря лучшему использованию оперативной памяти. Выбирать зигзагообразное или растущее вправо дерево имеет смысл, если отношения частично фрагментированы по непересекающимся домам и промежуточные отношения достаточно велики. В таком случае кустистые деревья обычно требуют больше стадий и выполняются дольше. С другой стороны, если промежуточные отношения малы, то конвейер не особенно эффективен, потому что трудно обеспечить сбалансированную нагрузку на все стадии конвейера.

Операторы в описанных выше деревьях должны улавливать параллелизм, что требует повторного секционирования входных отношений. Это можно продемонстрировать на примере алгоритма PHJ (см. раздел 8.4.1.2), где входные отношения секционируются с помощью применения одной и той же хеш-функции к атрибуту соединения, после чего производится параллельное соединение в локальных секциях. Чтобы упростить оптимизатору исследование пространства поиска, повторное секционирование данных можно инкапсулировать в *операторе обмена*. В зависимости от того, как производится секционирование, могут существовать различные операторы обмена, например: секционирование хешированием, секционирование по диапазонам или репликация данных на несколько узлов. Приведем примеры использования операторов обмена.

- Параллельное хеш-соединение: секционирование хешированием входных отношений по атрибуту соединения с последующим локальным соединением.
- Параллельное соединение методом вложенных циклов: репликация внутреннего отношения на те узлы, где секционировано внешнее отношение, с последующим локальным соединением.

- Параллельная сортировка диапазонов: секционирование по диапазонам с последующей локальной сортировкой.

На рис. 8.15 приведен пример дерева операторов с операторами обмена. Операция соединения производится с помощью секционирования хешированием входных отношений по атрибуту A (операторы $Xchg_1$ и $Xchg_2$) с последующим локальным соединением. Операции проецирования производятся путем устранения дубликатов в результате хеширования (оператор $Xchg_3$) с последующим локальным проецированием.

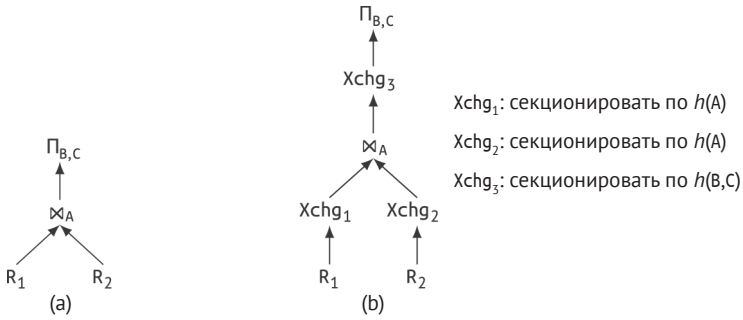


Рис. 8.15 ❖ Деревья с операторами обмена:
(a) последовательное дерево операторов; (b) параллельное дерево операторов

8.4.2.2. Модель стоимости

Напомним, что модель стоимости оптимизатора отвечает за оценку стоимости данного плана выполнения. Она состоит из двух частей: архитектурно-зависимой и архитектурно-независимой.

В архитектурно-независимую часть входят функции стоимости для алгоритмов реализации операторов, например метода вложенных циклов для соединения и последовательного доступа для выборки. Если оставить за скобками вопросы конкурентности, то в архитектурно-зависимую часть входят лишь функции стоимости для повторного секционирования данных и потребления памяти. Действительно, повторное секционирование кортежей отношения в системе без разделения ресурсов подразумевает передачу данных по межсоединению, тогда как в системах с общей памятью оно сводится к хешированию. Потребление памяти в системе без разделения ресурсов осложняется межоператорным параллелизмом. В системах же с общей памятью все операторы читают и записывают данные, пользуясь глобальной памятью, и легко проверить, хватит ли памяти для их параллельного выполнения, – для этого нужно, чтобы суммарное потребление памяти отдельными операторами было меньше доступной памяти. В системе без разделения ресурсов у каждого процессора своя память, поэтому важно знать, какие операторы выполняются параллельно на одном процессоре. Поэтому для простоты можно предполагать, что множества процессоров (дома), назначенные операторам, не перекрываются, т. е. либо их пересечение пусто, либо эти множества совпадают.

Полное время плана можно вычислить простым сложением затрат на всю процессорную обработку, ввод-вывод и передачу данных, как при оптимизации распределенных запросов. Вычислить время ответа сложнее, поскольку нужно учитывать конвейер.

Время ответа для плана p, разбитого на стадии (каждая обозначается ph), вычисляется по формуле:

$$RT(p) = \sum_{ph \in p} (\max_{Op \in ph}(respTime(Op) + pipe\_delay(Op)) + store\_delay(ph)),$$

где Op обозначает оператор, $respTime(Op)$ – время ответа Op, $pipe\_delay(Op)$ – период ожидания Op, необходимый, чтобы производитель доставил первые кортежи результата (равен 0, если входные отношения Op хранятся), $store\_delay(ph)$ – время, необходимое для сохранения результата стадии ph (равно 0, если ph – последняя стадия, в предположении, что результаты доставляются сразу после порождения).

Чтобы оценить стоимость плана выполнения, модель стоимости пользуется статистикой базы данных и информацией о ее организации, например о мощностях отношений и о секционировании, – точно так же, как при оптимизации распределенных запросов.

8.4.2.3. Стратегия поиска

Нет принципиальных причин, чтобы стратегия поиска отличалась от той, что применяется при централизованной или распределенной обработке запросов. Однако пространство поиска обычно гораздо больше, потому что на параллельные планы выполнения влияет больше параметров, в частности аннотации о конвейерном выполнении и сохранении отношений. Поэтому рандомизированные стратегии, например итеративное улучшение или имитация отжига, как правило, превосходят традиционные детерминированные стратегии при оптимизации параллельных запросов. Еще один интересный и при этом простой подход к уменьшению пространства поиска – двухэтапная стратегия оптимизации, предложенная для параллельной СУБД с общей памятью XPRS. Сначала на этапе компиляции порождается оптимальный план запроса, основанный на централизованной модели стоимости. Затем на этапе выполнения план распараллеливается с учетом таких динамически изменяющихся параметров, как доступный размер буфера и количество свободных процессоров. Показано, что этот подход почти всегда порождает оптимальные планы.

8.5. Балансировка запроса

Хорошая балансировка запроса критически важна для производительности параллельной системы. Время ответа множества параллельных операторов равно времени ответа самого долго выполняющегося. Поэтому требуется минимизировать самое длительное время ответа. Баланс нагрузки на различные узлы важен также для максимизации пропускной способности. Хотя

оптимизатор параллельных запросов включает решения о том, как выполнять параллельный план, сбалансированность нагрузки может ухудшиться из-за нескольких проблем, возможных во время выполнения. Решить эти проблемы можно на внутриоператорном и на межоператорном уровнях. В этом разделе мы обсудим как сами проблемы параллельного выполнения, так и их решения.

8.5.1. Проблемы параллельного выполнения

Основные проблемы, связанные с параллельным выполнением, – инициализация, интерференция и асимметрия.

Инициализация

Перед началом выполнения необходима инициализация. Этот шаг обычно последовательный и включает создание и инициализацию задач (или потоков), инициализацию взаимодействия и т. д. Его продолжительность пропорциональна степени параллелизма и для простых запросов, например выборки из одного отношения, может занимать основную часть времени выполнения. Таким образом, степень параллелизма следует выбирать в соответствии со сложностью запроса.

Можно вывести формулу для оценки максимального ускорения, достижимого при выполнении оператора, и из нее получить оптимальное число процессоров. Рассмотрим выполнение оператора, который обрабатывает N кортежей на n процессорах. Обозначим c среднее время обработки каждого кортежа, а a – время инициализации на одном процессоре. В идеальном случае время ответа при выполнении оператора равно

$$ResponseTime = (a * n) + \frac{c * N}{n}.$$

Отсюда можно получить оптимальное число процессоров n_{opt} и максимально достижимое ускорение ($Speed_{max}$):

$$n_{opt} = \sqrt{\frac{c * N}{a}}; \qquad Speed_{\max} = \frac{n_{opt}}{2}.$$

Оптимальное число процессоров (n_{opt}) не зависит от n, а определяется только полным временем обработки и временем инициализации. Поэтому максимизация степени параллелизма для одного оператора, например использования всех доступных процессоров, может негативно сказаться на ускорении из-за накладных расходов на инициализацию.

Интерференция

Хорошо распараллеленное выполнение может замедляться из-за *интерференции*, которая возникает, когда несколько процессоров одновременно обращаются к одному ресурсу, аппаратному или программному. Типичный пример аппаратной интерференции – состязание за шину межсоединения в системе с общей памятью. При увеличении количества процессоров возрас-

тает и количество конфликтов, что ограничивает возможности расширения системы. Решение состоит в том, чтобы дублировать общие ресурсы. Например, интерференцию при доступе к диску можно исключить путем добавления дополнительных дисков и секционирования отношений.

Программная интерференция возникает, когда несколько процессоров хотят получить доступ к разделяемым данным. Чтобы предотвратить рассогласованность, для защиты разделяемых данных используются переменные-мьютексы, которые позволяют разрешить доступ к данным только одному процессору, а остальные заблокировать. Это напоминает алгоритмы управления конкурентностью на основе блокировок (см. главу 5). Однако мьютексы вполне могут стать узким местом при выполнении запросов. Типичный пример программной интерференции – доступ к внутренним структурам базы данных, например индексам и буферам. Для простоты в первых СУБД они были защищены единственным мьютексом, за который возникала сильнейшая конкуренция.

Общее решение проблемы программной интерференции – расчленить разделяемый ресурс на несколько независимых ресурсов и защитить каждый из них своим мьютексом. Тогда к двум независимым ресурсам можно будет обращаться параллельно, что уменьшает вероятность интерференции. Чтобы еще снизить интерференцию при доступе к независимому ресурсу (например, индексной структуре), можно использовать репликацию. Доступ к реплицированным ресурсам тоже можно распараллелить.

Асимметрия

Проблемы балансировки нагрузки могут возникать как из-за внутриоператорного параллелизма (неодинаковые размеры секций) – это называется *асимметрией данных*, – так и из-за межоператорного параллелизма (неодинаковая сложность операторов).

Асимметричное распределение данных может по-разному влиять на параллельное выполнение. *Асимметрия значений атрибутов* (attribute value skew – *AVS*) внутренне присуща данным (например, в Париже живет больше людей, чем в Ватерлоо), а *асимметрия размещения кортежей* (tuple placement skew – *TPS*) возникает, когда данные изначально плохо секционированы (например, по диапазонам). *Асимметрия избирательности* (selectivity skew – *SS*) имеет место, когда избирательность предикатов выборки в разных узлах различна. *Асимметрия перераспределения* (Redistribution skew – *RS*) возникает на шаге перераспределения между двумя операторами, она похожа на TPS. Наконец, причиной *асимметрии соединения* (join product skew – JPS) является неодинаковость избирательности соединения в разных узлах. На рис. 8.16 эта классификация иллюстрируется на примере запроса с двумя неудачно секционированными отношениями `R` и `S`. Размер прямоугольников пропорционален размеру соответствующих секций. Такое плохое секционирование проистекает либо из асимметрии данных (AVS), либо из неудачно выбранной функции секционирования (TPS). При этом время обработки экземпляров `Scan1` и `Scan2` не одинаково. Случай оператора соединения еще хуже. Во-первых, количество полученных кортежей будет разным для разных экземпляров из-за плохого перераспределения секций `R` (RS) или неодинаковой избирательности в секциях `R` (SS). Наконец, из-за неодинако-

вых размеров секций S (AVS/TPS) время обработки кортежей, отправленных оператором последовательного просмотра, будет различаться, как и размер результата в одной и другой секции – из-за асимметричной избирательности соединения (JPS).

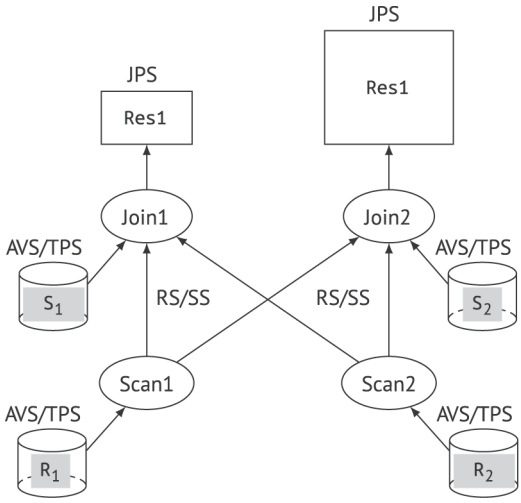


Рис. 8.16 ❖ Пример асимметрии данных

8.5.2. Внутриоператорная балансировка нагрузки

Качество внутриоператорной балансировки нагрузки зависит от степени параллелизма и выделения процессоров оператору. Для некоторых алгоритмов, например PHJ, эти параметры не ограничены размещением данных. Поэтому следует тщательно выбирать дом оператора (множество процессоров, которые его выполняют). Из-за проблемы асимметрии оптимизатору параллельных запросов трудно принять это решение статически (на этапе компиляции), поскольку для этого понадобилась бы очень точная и подробная модель стоимости. Поэтому основные подходы опираются на адаптивные или специализированные методы, которые можно включить в состав гибридного оптимизатора запросов. Мы опишем эти методы в контексте параллельной обработки соединения – темы, вызвавшей значительный интерес. Для простоты предположим, что дом каждого оператора определяется процессором запросов (статически или непосредственно перед началом выполнения).

Адаптивные методы

Идея заключается в том, чтобы статически назначить процессоры оператору (с использованием модели стоимости), а во время выполнения адаптироваться к асимметрии, перераспределив нагрузку. Последнее проще всего сделать, выявив слишком большие секции, разбить их на части и распределить между несколькими процессорами (из числа тех, что уже выделены для выполнения операции) для повышения степени параллелизма. Этот подход обобщается,

с тем чтобы более динамично подстраивать степень параллелизма. В план выполнения включаются особые *операторы контроля*, которые призваны обнаруживать отклонение статических оценок размеров промежуточных результатов от реальных значений во время выполнения. Если расхождение слишком велико, то оператор контроля перераспределяет отношение, чтобы предотвратить асимметрию соединения и асимметрию перераспределения. Адаптивные методы полезны для улучшения внутриоператорной балансировки нагрузки во всех параллельных архитектурах. Однако практическая работа по большей части выполнена для архитектур без разделения ресурсов, когда влияние несбалансированной нагрузки на производительность особенно велико. Пионером применения адаптивных методов, основанных на секционировании отношений (как в системах без разделения ресурсов), для архитектуры с общей памятью стала СУБД DBS3. Благодаря уменьшению интерференции процессоров этот метод обеспечивает отличную балансировку нагрузки для внутриоператорного параллелизма.

Специализированные методы

Параллельные алгоритмы соединения можно специализировать с учетом асимметрии. Один из таких подходов – использовать несколько алгоритмов соединения, каждый из которых специализирован для определенной степени асимметрии, и во время выполнения решать, какой алгоритм лучше. Эта идея опирается на две техники: секционирование по диапазонам и выборка. Секционирование по диапазонам используется вместо секционирования хешированием (в параллельном алгоритме хеш-соединения), чтобы избежать асимметрии перераспределения опорного отношения. Таким образом, процессоры могут получать секции, состоящие из одинакового числа кортежей, соответствующие различным диапазонам значений атрибута соединения. Чтобы найти границы диапазонов, производится выборка из опорного отношения и строится гистограмма значений атрибута соединения, т. е. количество кортежей для каждого значения этого атрибута. Выборка также помогает решить, какой взять алгоритм и какое отношение использовать для построения (опорное отношение), а какое – для апробирования. С помощью этих техник алгоритм параллельного хеш-соединения можно следующим образом адаптировать к асимметрии:

1) произвести выборку из опорного отношения для определения диапазонов секционирования;
2) перераспределить опорное отношение между процессорами, используя диапазоны. Каждый процессор строит хеш-таблицу, содержащую входящие кортежи;
3) перераспределить апробируемое отношение между процессорами, применяя те же диапазоны. Для каждого полученного кортежа каждый процессор апробирует хеш-таблицу для выполнения соединения.

Этот алгоритм борьбы с асимметрией можно еще улучшить, применяя дополнительные приемы и другие стратегии выделения процессоров. Похожий подход заключается в том, чтобы вставить в алгоритм соединения шаг диспетчеризации, отвечающий за перераспределение нагрузки во время выполнения.

8.5.3. Межоператорная балансировка нагрузки

Чтобы добиться хорошего баланса нагрузки на межоператорном уровне, необходимо для каждого оператора решить, сколько и каких процессоров выделить для его выполнения. При этом нужно учитывать конвейерный параллелизм, для которого требуется межоператорное взаимодействие. В системах без разделения ресурсов это труднее по следующим причинам. Во-первых, решения о степени параллелизма и выделении процессоров операторам, принимаемые на этапе оптимизации распараллеливания, основаны на потенциально неточной модели стоимости. Во-вторых, решение о степени параллелизма может оказаться ошибочным из-за того, что процессоры и операторы – дискретные сущности. Наконец, процессоры, выделенные последним операторам в конвейере, могут долгое время простаивать. Это называется проблемой задержки в конвейере.

В системах без разделения ресурсов основной подход – принимать решение о степени параллелизма и назначении процессоров каждому оператору динамически (непосредственно перед началом выполнения). Например, в алгоритме согласования темпов модель стоимости используется, чтобы согласовать темпы производства и потребления кортежей. На этой основе выбирается множество процессоров, которые будут задействованы в выполнении запроса (зная размер доступной памяти, количество доступных процессоров и характер использования диска). Есть и много других алгоритмов для выбора количества и мест расположения процессоров, например максимизация использования нескольких ресурсов на основе статистики работы с ними.

В архитектурах с общей памятью или диском гибкости больше, потому что у всех процессоров имеется равный доступ к дискам. Поскольку нет необходимости в физическом секционировании отношения, каждый процессор может быть назначен любому оператору. В частности, процессор может быть назначен всем операторам в одной и той же конвейерной цепочке, т. е. межоператорный параллелизм отсутствует. Однако межоператорный параллелизм полезен для выполнения независимых конвейерных цепочек. Подход, предложенный в системе с общей памятью XPRS, допускает параллельное выполнение независимых конвейерных цепочек, называемых задачами. Основная идея заключается в том, чтобы сочетать задачи, ограниченные скоростью ввода-вывода, и задачи, ограниченные мощностью процессора, для оптимального использования системных ресурсов. До начала выполнения задача следующим образом – с применением модели стоимости – классифицируется как ограниченная вводом-выводом или мощностью процессора. Предположим, что при последовательном выполнении задача t обращается к диску с частотой $IO_{rate}(t)$, измеряемой количеством операций доступа к диску в секунду. Рассмотрим систему с общей памятью с n процессорами и полной пропускной способностью диска B (количество операций доступа к диску в секунду). Задача t считается ограниченной скоростью ввода-вывода, если $IO_{rate}(t) > B/n$, в противном случае она ограничена мощностью процессора. Ели задачи обоих типов выполнять параллельно, то баланс между вводом-

выводом и использование процессора будут оптимальными. Это достигается динамическим подстраиванием внутриоператорного параллелизма в задачах с целью достижения максимального использования ресурсов.

8.5.4. Внутризапросная балансировка нагрузки

Для внутризапросной балансировки нагрузки необходимо сочетать внутри- и межоператорный параллелизм. До некоторой степени, зависящей от параллельной архитектуры, описанные выше методы внутри- и межоператорной балансировки нагрузки можно комбинировать. Но в кластерах без разделения ресурсов, узлы которых имеют общую память (или многоядерные процессоры), проблемы балансировки нагрузки обостряются, поскольку их необходимо решать на двух уровнях: локально – между процессорами или ядрами каждого узла с общей памятью и глобально – между всеми узлами. Ни один из ранее рассмотренных подходов к внутри- и межоператорной балансировке нагрузки прямо не обобщается на эту задачу. Стратегии балансировки нагрузки в системах без разделения ресурсов сталкиваются с еще более серьезными проблемами (например, сложность и неточность модели стоимости). С другой стороны, адаптация динамических решений, разработанных для систем с общей памятью, потребовала бы очень высоких затрат на передачу данных.

Общее решение проблемы балансировки нагрузки – модель выполнения, называемая *динамической обработкой* (*ДО*). Основная ее идея заключается в том, чтобы разложить запрос на автономные единицы последовательной обработки, каждую из которых можно поручить одному процессору. Интуитивно понятно, что процессор может мигрировать по горизонтали (внутриоператорный параллелизм) и по вертикали (межоператорный параллелизм) вдоль множества операторов запроса. Это минимизирует накладные расходы на передачу данных при межузловой балансировке нагрузки путем усиленной внутри- и межоператорной балансировки нагрузки в пределах узлов с общей памятью. На вход модели выполнения подается план параллельного выполнения, порожденный оптимизатором, т. е. дерево операторов с диспетчеризацией и выделением вычислительных ресурсов операторам. Ограничения на диспетчеризацию операторов выражают частичный порядок на множестве операторов запроса: $Op_1 \prec Op_2$ означает, что оператор Op_1 не может начаться раньше, чем Op_2.

Пример 8.5. На рис. 8.17 показано дерево соединений с четырьмя отношениями R_1, R_2, R_3 и R_4 и соответствующее ему дерево операторов с четко выделенными конвейерными цепочками. В предположении, что используется параллельное хеш-соединение, существуют следующие ограничения диспетчеризации между связанными операторами построения и апробирования:

$Build1 \prec Probe1$
$Build2 \prec Probe3$
$Build3 \prec Probe2$

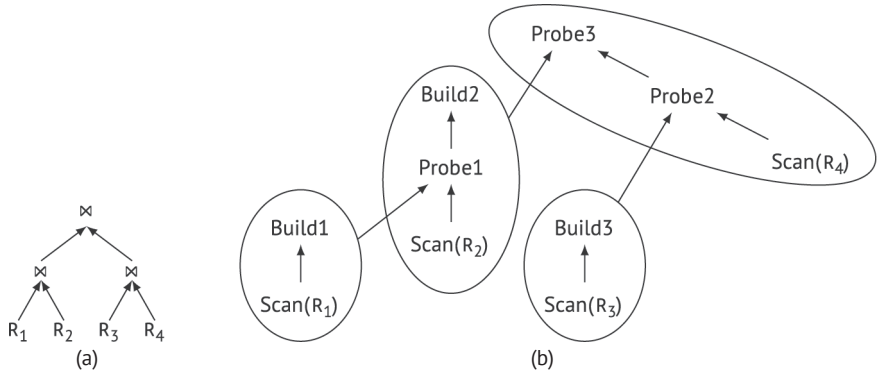


Рис. 8.17 ❖ Дерево соединений и ассоциированное с ним дерево операторов: (a) дерево соединений; (b) дерево операторов (овалами представлены конвейерные цепочки)

Существуют также эвристики диспетчеризации между операторами из различных конвейерных цепочек, вытекающие из следующих ограничений диспетчеризации:

Эвристика1: Build1 ≺ Scan(R_2), Build3 ≺ Scan(R_4), Build2 ≺ Scan(R_3)
Эвристика2: Build2 ≺ Scan(R_3)

В предположении, что имеется три узла с общей памятью (ОП-узла) *i*, *j* и *k*, причем R_1 хранится в узле *i*, R_2 и R_3 – в узле *j*, а R_4 – в узле *k*, имеем следующие дома операторов:

дом(Scan(R_1)) = *i*
дом(Build1, Probe1, Scan(R_2), Scan(R_3)) = *j*
дом(Scan(R_4)) = *k*
дом(Build2, Build3, Probe2, Probe3) = *j* и *k* ♦

При таком дереве операторов задача заключается в том, чтобы найти способ выполнения, минимизирующий время ответа. Это можно сделать, воспользовавшись механизмом динамической балансировки нагрузки на двух уровнях: (i) внутри ОП-узла балансировка достигается с помощью быстрого межпроцессного взаимодействия; (ii) между ОП-узлами требуется более дорогая передача сообщений. Таким образом, проблема сводится к построению такой модели выполнения, при которой максимизируется использование локальной балансировки нагрузки и минимизируется использование глобальной (посредством передачи сообщений).

Назовем *активацией* наименьшую единицу последовательной обработки, не допускающую дальнейшего секционирования. Основное свойство модели ДО в том, что она позволяет любому процессору обработать любую активацию в его собственном ОП-узле. Таким образом, между потоками и операторами не существует статической ассоциации. Это обеспечивает хороший баланс нагрузки для внутри- и межоператорного параллелизма в пределах

ОП-узла, а значит, сводит к минимуму потребность в глобальной балансировке нагрузки, т. е. возникновение ситуаций, когда ОП-узлу нечего делать.

ДО-модель выполнения основана на нескольких понятиях: активация, очередь активаций и потоки.

Активации

Активация представляет последовательную единицу работы. Поскольку любая активация может быть выполнена любым потоком (любым процессором), активации должны быть автономными и содержать ссылки на все необходимое для выполнения: исполняемый код и подлежащие обработке данные. Можно выделить два вида активаций: по триггеру и по данным. *Активация по триггеру* применятся, чтобы начать выполнение листового оператора, т. е. последовательного просмотра. Она представлена парой (*Оператор*, *Секция*), состоящей из оператора последовательного просмотра и просматриваемой секции базового отношения. *Активация по данным* описывает кортеж, порожденный в конвейерном режиме. Она представлена тройкой (*Оператор*, *Кортеж*, *Секция*), которая ссылается на подлежащий выполнению оператор. Для оператора построения активация по данным говорит, что кортеж должен быть вставлен в хеш-таблицу корзины, а для оператора апробирования – что кортеж должен быть апробирован хеш-таблицей секции. Хотя активации автономны, их можно выполнять только в ОП-узле, где хранятся ассоциированные данные (хеш-таблицы или базовые отношения).

Очереди активаций

Перемещение активации по данным вдоль конвейерных цепочек производится с помощью *очередей активаций*, ассоциированных с операторами. Если производитель и потребитель активации находятся в одном и том же ОП-узле, то перемещение осуществляется в общей памяти. В противном случае необходима передача сообщений. Для унификации модели выполнения очереди используются как для активаций по триггерам (подаваемых на вход операторов последовательного просмотра), так и для активаций по кортежам (подаваемых на вход операторов построения и апробирования). Любой поток имеет неограниченный доступ ко всем очередям, находящимся в его ОП-узле. Управление небольшим количеством очередей (например, по одной для каждого оператора) может приводить к интерференции. Для ее уменьшения одна очередь ассоциируется с каждым потоком, исполняющим оператор. Заметим, что при увеличении количества очередей интерференция уменьшается, но ценой увеличения накладных расходов на управление очередями. Чтобы снизить интерференцию, не увеличивая количество очередей, каждому потоку предоставляется приоритетный доступ ко множеству очередей, называемых его *первичными очередями*. Таким образом, поток сначала пытается потребить активации в своих первичных очередях. Во время выполнения может оказаться, что в силу ограничений на диспетчеризацию операторов некоторый оператор блокируется до завершения каких-то других операторов (блокирующих его). Поэтому очередь к заблокированному оператору также блокируется, т. е. находящиеся в ней активации нельзя потреблять, но добавлять активации в нее можно, если производящий оператор не за-

блокирован. Когда все блокирующие операторы завершатся, потребление из ранее заблокированной очереди возобновляется. Это показано на рис. 8.18, где представлен моментальный снимок дерева операторов на рис. 8.17.

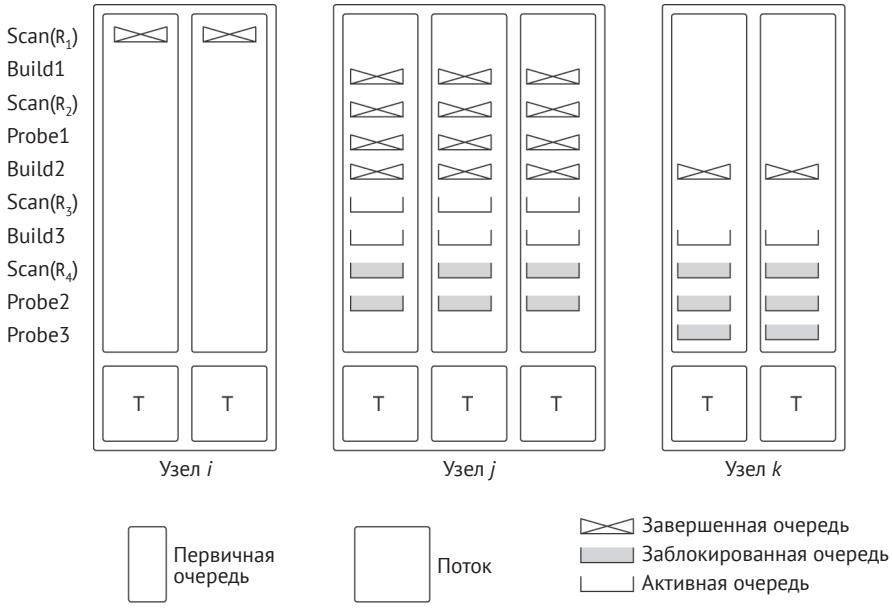


Рис. 8.18 ❖ Моментальный снимок выполнения

Потоки

Простая стратегия достижения хорошо сбалансированной нагрузки внутри ОП-узла заключается в том, чтобы создать гораздо больше потоков, чем имеется процессоров, и позволить операционной системе заняться их планированием. Однако при такой стратегии резко возрастает количество системных вызовов, необходимых для планирования потоков, а также интерференция между потоками. Вместо того чтобы поручать балансировку нагрузки операционной системе, можно выделить только один поток на каждый процессор для обработки одного запроса. Это возможно, потому что любой поток может выполнить любой оператор, назначенный его ОП-узлу. Преимущество стратегии «один процессор – один поток» в том, что значительно снижаются затраты на синхронизацию и преодоление интерференции, при условии что поток никогда не блокируется.

Балансировка нагрузки в пределах ОП-узла достигается за счет того, что все очереди активаций выделяются в сегменте общей памяти и всем потокам разрешается потреблять активации из любой очереди. Чтобы ограничить интерференцию, поток потребляет столько, сколько может, из множества своих первичных очередей и только потом рассматривает другие очереди в своем ОП-узле. Поэтому поток переходит в состояние простоя, только когда не осталось активаций ни для какого оператора, а это значит, что в ОП-узле больше нет работы.

Если ОП-узел остался без работы, мы снижаем нагрузку на какой-то другой ОП-узел, заимствуя часть назначенной ему работы. Однако заимствование активаций (путем передачи сообщений) сопряжено с накладными расходами на передачу данных. Кроме того, одного лишь заимствования активаций недостаточно – нужно еще получить ассоциированные с ними данные, т. е. хеш-таблицы. Поэтому выигрыш от заимствования активаций и данных необходимо динамически оценивать.

Объем балансировки нагрузки зависит от количества конкурентно выполняемых операторов, что открывает возможность для поиска какой-нибудь работы во время простоя. Чтобы увеличить количество конкурентных операторов, можно разрешить выполнение нескольких конвейерных цепочек или воспользоваться неблокирующими алгоритмами хеш-соединения, которые допускают конкурентное выполнение всех операторов в кустистом дереве. С другой стороны, при конкурентном выполнении большого числа операторов возрастает потребление памяти. Статическая диспетчеризация операторов, осуществляемая оптимизатором, должна избегать переполнения памяти и находить приемлемый компромисс.

8.6. Отказоустойчивость

В данном разделе мы обсудим, что происходит в случае отказов. В этой связи возникает несколько вопросов. Во-первых, как поддержать согласованность, несмотря на отказы. Во-вторых, как отработать отказ для еще не выполненных транзакций. В-третьих, когда реплика вновь становится доступной (после восстановления) или в системе появляется новая реплика, необходимо перестроить текущее состояние базы данных. Главная проблема – как быть с отказами. Прежде всего отказ необходимо обнаружить. В подходах на основе групповой коммуникации (см. главу 6) отказ обнаруживается благодаря базовому механизму групповой коммуникации (обычно основанному на тех или иных контрольных сигналах). Уведомления об изменении состава членов посылаются в виде событий[1]. Сравнив новый состав членов с предыдущим, мы легко узнаем, какие реплики вышли из строя. Механизм групповой коммуникации также гарантирует, что представление всех подключенных реплик о составе членов одинаково. Если механизма групповой коммуникации нет, то обнаружение отказов можно либо поручить уровню передачи данных (например, TCP/IP), либо реализовать в виде дополнительной компоненты логики репликации. Но необходим какой-то протокол, гарантирующий, что все подключенные реплики одинаково представляют себе, какие реплики работают, а какие нет. В противном случае возникнет рассогласование.

Отказы необходимо также обнаруживать на стороне клиента с помощью клиентского API. Обычно клиенты подключаются по протоколу TCP/IP и могут

---

[1] В литературе по групповой коммуникации для обозначения события, содержащего информацию об изменении состава членов, употребляется термин *изменение представления*. Здесь мы не пользуемся им, чтобы не путаться с понятием *представления* в базе данных.

заподозрить отказ узла, если соединение разрывается. После отказа реплики клиентский API должен найти другую реплику, подключиться к ней и в простейшем случае повторно передать последнюю невыполненную транзакцию этой реплике. Поскольку производится повторная передача, возможна доставка дубликатов. Поэтому необходим механизм обнаружения и устранения дубликатов. В большинстве случаев достаточно, чтобы у клиента был уникальный идентификатор и каждая транзакция на стороне клиента тоже имела уникальный идентификатор. Последняя задача решается путем увеличения идентификатора на единицу для каждой новой транзакции. Таким образом, кластер должен следить за тем, была ли уже обработана клиентская транзакция и, если да, отбросить ее.

После обнаружения отказавшей реплики необходимо предпринять несколько действий. Все они являются частью процесса отработки отказа, который должен перенаправить транзакции с отказавшего узла на узел другой реплики, так чтобы это было прозрачно для клиентов. Способ отработки отказа зависит от того, была ли отказавшая реплика главной. Если вышла из строя неглавная реплика, то со стороны кластера не требуется никаких действий. Клиенты, у которых имеются невыполненные транзакции, подключаются к узлу другой реплики и заново отправляют ему транзакции. Но возникает интересный вопрос: как определена согласованность? Напомним (см. раздел 6.1), что в реплицированной базе данных «сериализуемость как в одной копии» может быть нарушена в результате сериализации транзакций в разных узлах в разном порядке. Из-за отработки отказа транзакции тоже могут быть обработаны таким образом, что сериализуемость как в одной копии нарушается.

В большинстве подходов к репликации отработка отказа сводится к отмене всех выполняющихся транзакций, чтобы предотвратить такое развитие событий. Но это затрагивает клиентов, которые должны перезапустить отмененные транзакции. Поскольку клиент обычно не располагает транзакционными возможностями, чтобы откатить результаты интерактивного взаимодействия, это может оказаться очень трудным делом. Благодаря концепции *высокодоступных транзакций* удается сделать отказы полностью прозрачными для клиентов, так что им не нужно готовиться к отмене транзакций из-за отказов.

Действия в случае отказа главной реплики сложнее, поскольку необходимо назначить новую главную реплику взамен отказавшей. Назначение новой главной реплики должно быть согласовано со всеми репликами в кластере. В случае репликации, основанной на групповой коммуникации, приходит уведомление об изменении состава членов, поэтому для назначения главной реплики достаточно применить детерминированную функцию (все узлы получают в точности один тот же список подключенных узлов).

Еще одна важная сторона отказоустойчивости – восстановление после отказа. Для обеспечения высокой доступности необходимо справляться с отказами и предоставлять согласованный доступ к данным, несмотря на отказы. Однако отказ уменьшает степень избыточности в системе, а значит, снижает доступность и производительность. Поэтому необходимо заново вводить в строй отказавшие или новые реплики, чтобы поддержать

или даже повысить уровень доступности и производительности. Основная трудность заключается в том, что реплики обладают состоянием, а во время простоя отказавшей реплики некоторые обновления могли быть пропущены. Поэтому в процессе восстановления отказавшая реплика должна получить все пропущенные обновления, прежде чем сможет приступить к обработке новых транзакций. Решение состоит в том, чтобы прекратить обработку транзакций. Поэтому система переходит в состояние покоя, о чем любая работающая реплика может уведомить восстанавливающуюся. После того как восстанавливающаяся реплика получит все пропущенные обновления, обработку транзакций можно возобновить, и все реплики смогут обрабатывать новые транзакции.

8.7. Кластеры баз данных

В параллельной системе баз данных функции параллельного управления данными обычно реализованы сильно связанным образом, когда все однородные узлы находятся под полным контролем параллельной СУБД. Более простое (но не столь эффективное) решение – воспользоваться *кластером баз данных*, т. е. группой автономных баз данных, каждая из которых управляется своим экземпляром стандартной СУБД. Основное отличие от параллельной СУБД, реализованной в кластере, заключается в использовании в каждом узле СУБД, рассматриваемой как черный ящик. Поскольку исходный код СУБД зачастую недоступен и модифицировать его в расчете на работу в кластере невозможно, средства параллельного управления данными приходится реализовывать с помощью ПО промежуточного уровня. Этот подход успешно применен в кластерах на основе MySQL и PostgreSQL.

Было проведено немало исследований, направленных на то, чтобы в полной мере воспользоваться кластерной средой (с ее надежным и быстрым механизмом коммуникации) с целью повышения производительности и доступности за счет репликации данных. Главными результатами этих исследований стали новые методы репликации, балансировки нагрузки и обработки запросов. В данном разделе мы опишем эти методы, но сначала познакомимся с архитектурой кластера баз данных.

8.7.1. Архитектура кластера баз данных

На рис. 8.19 показан кластер баз данных с архитектурой без разделения ресурсов. Параллельное управление данными реализуется независимыми СУБД, координируемыми ПО промежуточного уровня, которое размещено в каждом узле. Чтобы улучшить производительность и доступность, данные можно реплицировать в разных узлах с помощью локальных СУБД. Клиентские приложения взаимодействуют с ПО промежуточного уровня классическим способом – отправляя транзакции, т. е. ситуативные запросы, транзакции или обращения к хранимым процедурам. Некоторые узлы могут играть

специальную роль узлов доступа для приема транзакций, тогда они пользуются общей глобальной службой каталогов, в которой хранится информация о пользователях и базах данных. Обработка транзакции, адресованной одной базе данных, производится следующим образом. Сначала транзакция аутентифицируется и авторизуется с помощью службы каталогов. Если все хорошо, транзакция отправляется СУБД в каком-то, возможно, другом узле для выполнения. В разделе 8.7.4 мы увидим, как эту модель можно обобщить на параллельную обработку запросов с использованием нескольких узлов для обработки одного запроса.

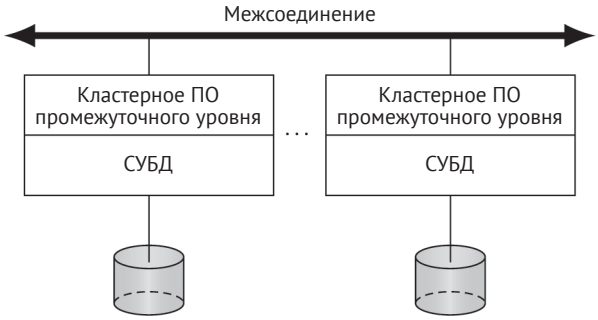


Рис. 8.19 ❖ Кластер баз данных без разделения ресурсов

Как и параллельные СУБД, промежуточное ПО в кластере баз данных состоит из нескольких программных уровней: балансировщик транзакционной нагрузки, диспетчер репликации, процессор запросов и диспетчер отказоустойчивости. Балансировщик транзакционной нагрузки запускает выполнение транзакции в оптимальном узле, используя информацию о нагрузке, полученную с помощью пробных запросов. «Оптимальным» считается узел с наименьшей транзакционной нагрузкой. Кроме того, балансировщик транзакционной нагрузки гарантирует, что при выполнении любой транзакции соблюдаются свойства ACID, а затем просит СУБД зафиксировать или отменить транзакцию. Диспетчер репликации управляет доступом к реплицированным данным и обеспечивает сильную согласованность таким образом, что транзакции, которые обновляют реплицированные данные, выполняются в одном и том же порядке во всех узлах. Процессор запросов задействует внутри- и межзапросный параллелизм. Что касается межзапросного параллелизма, то процессор запросов отправляет каждый полученный запрос одному узлу и после его завершения посылает результаты клиентскому приложению. С внутризапросным параллелизмом дело обстоит несколько сложнее. Поскольку автономные СУБД ничего не знают о кластере, они не могут взаимодействовать в ходе обработки одного запроса. Поэтому управление выполнением запроса, формирование окончательного результата и балансировка нагрузки ложатся на процессор запросов. Наконец, диспетчер отказоустойчивости отвечает за оперативное восстановление и отработку отказов.

8.7.2. Репликация

Как и в распределенных СУБД, репликацию можно использовать для повышения производительности и доступности. В кластере баз данных быстрой системой межсоединения и коммуникации можно воспользоваться для поддержки сериализуемости как в одной копии, одновременно обеспечив масштабируемость (для повышения производительности с увеличением количества узлов) и автономность (для работы с СУБД, рассматриваемыми как черный ящик). Кластер предоставляет стабильную среду со слабо изменяющейся топологией (например, в результате добавления узлов или отказа линии связи). Таким образом, он упрощает поддержку системы групповой коммуникации, которая в ответе за надежное взаимодействие групп узлов. Примитивы групповой коммуникации (см. раздел 6.4) можно использовать совместно с энергичной или ленивой репликацией как средство атомарного распространения информации (вместо дорогого протокола 2PC).

Теперь мы представим еще один протокол – *превентивную репликацию*, который является ленивым и поддерживает сериализуемость как в одной копии и масштабируемость. Кроме того, превентивная репликация сохраняет автономность СУБД. Вместо вполне упорядоченной многоадресной передачи в этом протоколе применяется надежная многоадресная передача с помощью FIFO-очереди, которая проще и эффективнее. Принцип ее таков. Каждая транзакция T, поступающая в систему, имеет хронологическую временную метку $ts(T) = C$ и передается всем узлам, где хранятся копии. В каждом узле выполнению T предшествует некоторая временная задержка, равная верхней границе времени, необходимого для рассылки сообщения по нескольким адресам (предполагается, что система синхронная с ограниченным временем вычисления и передачи данных). Критически важно точно вычислять эти верхние границы для сообщений (т. е. задержку). Впрочем, в кластерной системе это можно сделать весьма точно. По истечении времени задержки есть гарантия, что все транзакции, которые могли бы быть зафиксированы до момента C, уже получены и выполнены раньше T в порядке следования временных меток (т. е. вполне упорядоченно). Таким образом, этот подход предотвращает конфликты и гарантирует сильную согласованность кластера баз данных. Введение временных задержек использовалось в нескольких ленивых централизованных протоколах репликации для распределенных систем. Испытания протокола превентивной репликации в эталонных тестах TPC-C в кластере из 64 узлов под управлением СУБД PostgreSQL показали отличную вертикальную масштабируемость и ускорение работы.

8.7.3. Балансировка нагрузки

В кластере баз данных репликация открывает хорошие возможности для балансировки нагрузки. Протоколы энергичной или превентивной репликации (см. раздел 8.7.2) позволяют легко обеспечить балансировку. Поскольку все копии взаимно согласованы, любой узел, где хранится копия данных транзакции, например наименее нагруженный, можно выбрать во время выпол-

нения, применив традиционную стратегию балансировки нагрузки. Балансировка транзакционной нагрузки также не вызывает сложностей в случае ленивой распределенной репликации, поскольку все главные узлы должны в конечном счете выполнить транзакцию. Однако полная стоимость выполнения транзакции во всех узлах может оказаться высокой. Если ослабить определение согласованности, то ленивая репликация может уменьшить стоимость выполнения транзакций и тем самым повысить производительность запросов и транзакций. Таким образом, в зависимости от требований к согласованности и производительности в кластерах баз данных могут найти применение как энергичные, так и ленивые протоколы репликации.

8.7.4. Обработка запросов

В кластере баз данных для повышения производительности можно с успехом использовать параллельную обработку запросов. Межзапросный параллелизм является естественным результатом балансировки нагрузки и репликации, о чем было сказано в предыдущем разделе. Такой параллелизм полезен прежде всего для увеличения пропускной способности в транзакционных приложениях и в какой-то мере позволяет уменьшить время ответа транзакций и запросов. Для OLAP-приложений, в которых часто встречаются ситуативные запросы, обращающиеся к большому количеству данных, для снижения времени ответа важен также внутризапросный параллелизм, когда один запрос обрабатывается в разных секциях участвующих в нем отношений.

Есть два способа секционирования отношения в кластере баз данных: физическое и виртуальное. При физическом секционировании определяются секции отношения, по сути дела горизонтальные фрагменты, которые затем размещаются в узлах кластера, возможно, с репликацией. Это похоже на проектирование фрагментации и размещения в распределенных базах данных (см. главу 2), но цель состоит в том, чтобы повысить степень внутризапросного параллелизма, а не локальность ссылок. Поэтому секции должны быть гораздо мельче, хотя точные оценки зависят от размера отношений и запроса. При физическом секционировании в кластерах баз данных, созданных для поддержки принятия решений, можно использовать фрагменты малого размера. Показано, что при равномерном распределении данных это решение дает хороший внутризапросный параллелизм и по производительности превосходит межзапросный параллелизм. Однако физическое секционирование статично и потому чувствительно к асимметрии данных и изменению типов запросов – в таких случаях может понадобиться периодическая рефрагментация.

Виртуальное секционирование позволяет обойти проблемы физического секционирования с помощью динамического подхода и полной репликации (каждое отношение реплицируется на все узлы). В простейшей форме, которую мы будем называть *простым виртуальным секционированием* (*ПВС*), виртуальные секции динамически порождаются для каждого запроса, а внутризапросный параллелизм достигается путем отправки подзапросов разным виртуальным секциям. Для порождения подзапросов процессор запросов

в кластере баз данных добавляет к входному запросу предикаты, чтобы ограничить доступ подмножеством отношения, т. е. виртуальной секцией. Он может также прибегнуть к переписыванию запроса, чтобы разложить исходный запрос на эквивалентные подзапросы, а затем собрать их воедино с помощью композиционного запроса. Тогда каждая СУБД, получившая подзапрос, должна будет обработать разные подмножества данных. И на последнем этапе результаты подзапросов объединяются агрегирующим запросом.

Пример 8.6. Проиллюстрируем ПВС на примере следующего запроса *Q*:

```
SELECT PNO, AVG(DUR)
FROM   WORKS
WHERE  SUM(DUR) > 200
GROUP BY PNO
```

Запрос к виртуальной секции получается добавлением во фразу `WHERE` запроса *Q* предиката «`and PNO >= 'P1' and PNO < 'P2'`». Привязывая `['P1', 'P2']` к *n* последовательным диапазонам значений `PNO`, мы получаем *n* подзапросов, по одному для всех виртуальных секций `WORKS`. Таким образом, степень внутризапросного параллелизма равна *n*. Дополнительно операцию **`AVG`**`(DUR)` в подзапросе следует переписать в виде **`SUM`**`(DUR)`, **`COUNT`**`(DUR)`. Наконец, для получения правильного значения **`AVG`**`(DUR)` необходим композиционный запрос, в котором выполняется операция **`SUM`**`(DUR)/`**`SUM`**`(`**`COUNT`**`(DUR))` для *n* частичных результатов.

Производительность выполнения каждого подзапроса сильно зависит от методов доступа к атрибуту, по которому производится секционирование (`PNO`). В данном примере лучше всего был бы кластерный индекс по `PNO`. Таким образом, процессору запросов важно знать об имеющихся методах доступа, чтобы решить, по какому атрибуту производить секционирование для данного запроса. ♦

ПВС допускает значительную гибкость выделения узлов во время обработки запроса, поскольку для выполнения подзапроса может быть выбран любой узел. Однако не все виды запросов можно распараллелить с помощью ПВС. Мы можем разбить все множество OLAP-запросов на классы со схожими свойствами распараллеливания. В основу классификации положена информация о том, как производится доступ к самым большим отношениям, которые в типичном OLAP-приложении называются таблицами фактов. Идея в том, что виртуальное секционирование таких отношений дает наибольший внутриоператорный параллелизм. Можно выделить три основных класса:

1) запросы, в которых нет подзапросов, обращающихся к таблице фактов;
2) запросы, в которых есть подзапрос, эквивалентный запросу класса 1;
3) все остальные запросы.

Для применения ПВС запросы класса 2 необходимо переписать в запросы класса 1, а запросы класса 3 вообще не получают выигрыша от использования ПВС.

У ПВС есть некоторые ограничения. Во-первых, найти наилучший атрибут для виртуального секционирования и диапазоны его значений не всегда легко, потому что предположение о равномерном распределении значений

редко оправдывается на практике. Во-вторых, некоторые СУБД предпочитают полный просмотр таблицы вместо доступа по индексу, когда извлекаются кортежи со значениями из больших интервалов. Это уменьшает выигрыш от параллельного доступа к диску, поскольку одному узлу для доступа к виртуальной секции может потребоваться прочитать все отношение. Поэтому ПВС оказывается зависимым от оптимизатора запросов в СУБД, установленной в узлах. В-третьих, поскольку запрос невозможно модифицировать извне во время выполнения, балансировку нагрузки трудно обеспечить, и она зависит от начального секционирования.

Виртуальное секционирование с мелкими секциями обходит эти ограничения благодаря использованию большого числа подзапросов вместо одного на каждую СУБД. Работа с меньшими подзапросами позволяет избежать полного просмотра таблиц и делает обработку запросов менее чувствительной к особенностям СУБД. Однако при таком подходе необходимо оценивать размеры секций, опираясь на статистику базы данных и оценки времени выполнения запросов. На практике получить такие оценки, рассматривая СУБД как черный ящик, затруднительно.

Адаптивное виртуальное секционирование (*ABC*) решает эту проблему, динамически настраивая размеры секций, что не требует таких оценок. ABC работает независимо в каждом участвующем узле кластера, обходясь без межузлового взаимодействия (для определения размера секций). Вначале каждый узел получает интервал значений, с которым может работать. Эти интервалы определяются точно так же, как в ПВС. Затем каждый узел выполняет следующие действия:

1) начать с очень малого размера секции, начинающейся с первого значения из полученного интервала;
2) выполнить подзапрос с таким интервалом;
3) увеличить размер секции и выполнять соответствующий подзапрос, пока время выполнения растет медленнее, чем размер секции;
4) прекратить увеличение. Найден стабильный размер;
5) если имеется место падения производительности, т. е. время выполнения становится хуже, уменьшить размер и перейти к шагу 2.

Начиная с очень малого размера секции, мы избегаем полного просмотра таблицы в начале процесса. Также нам не нужно знать заранее, после какого порога СУБД перестает использовать кластерные индексы и начинает просматривать таблицу целиком. По мере увеличения размера секции мы следим за временем выполнения, что позволяет определить точку, после которой время обработки запроса становится сильно зависимым от размера данных. Например, если при удвоении размера секции время выполнения тоже удваивается, значит, такая точка найдена. В результате алгоритм перестает увеличивать размер. Производительность системы может снизиться из-за непопадания в кеш данных СУБД или из-за общего возрастания нагрузки на систему. Может случиться, что используемый размер слишком велик, а выигрыш достигался просто потому, что данные уже были в кеше. В таком случае размер лучше уменьшить. Именно для этого и предназначен шаг 5. Он дает шанс вернуться и проанализировать, как обстоит дело с меньшим размером секции. С другой стороны, если падение производительности объ-

ясняется спонтанным и временным увеличением нагрузки на систему или непопаданием в кеш данных, то сохранение небольшого размера секции может ухудшить дело. Чтобы избежать такой ситуации, алгоритм возвращается к шагу 2 и снова начинает увеличивать размер.

У ABC и других вариантов виртуального секционирования есть несколько преимуществ: гибкость выделения узлов, высокая доступность благодаря полной репликации и возможности для динамической балансировки нагрузки. Но полная репликация может увеличить количество потребного места на дисках. Для поддержки гибридной репликации были предложены решения, объединяющие физическое и виртуальное секционирование. В этом случае физическое секционирование применяется для самых больших и важных отношений, а мелкие таблицы реплицируются полностью. Таким образом, внутризапросного параллелизма можно достичь при меньших требованиях к емкости диска. В гибридном секционировании решение ABC сочетается с физическим секционированием. При этом решается проблема места на диске и сохраняются преимущества ABC, т. е. предотвращение полного просмотра таблиц и динамическая балансировка нагрузки.

8.8. Резюме

В параллельных системах баз данных используются многопроцессорные архитектуры для обеспечения высокой производительности, доступности, расширяемости и масштабируемости с хорошим отношением стоимость–производительность. Кроме того, параллелизм – единственно возможное практическое решение для поддержки очень больших баз данных и приложений в одной системе.

Архитектуры параллельных баз данных можно подразделить на три категории: с общей памятью, с общим диском и без разделения ресурсов. У каждой из них есть свои плюсы и минусы. Архитектура с общей памятью используется в сильно связанных NUMA-мультипроцессорах или многоядерных процессорах, она характеризуется наивысшей производительностью в силу быстрого доступа к памяти и отличной сбалансированности нагрузки. Архитектуры с общим диском и без разделения ресурсов применяются в компьютерных кластерах, состоящих чаще всего из многоядерных процессоров. Будучи дополнены сетями с малой задержкой (например, Infiniband или Myrinet), они могут обеспечить высокую производительность и масштабируются на очень большие конфигурации (с тысячами узлов). Кроме того, возможности технологии RDMA в этих сетях могут быть использованы для построения экономически эффективных NUMA-кластеров. Архитектура с общим диском часто используется в системах с рабочей нагрузкой типа OLTP, поскольку она проще и позволяет добиться хорошей балансировки нагрузки. Однако архитектура без разделения ресурсов остается единственным выбором для высокомасштабируемых систем с хорошим отношением стоимость–производительность, какие нужны для OLAP-приложений или обработки больших данных.

Методы параллельного управления данными являются обобщением методов, применяемых в распределенных базах данных. Однако для таких архитектур критически важны вопросы секционирования данных, репликации, параллельной обработки запросов, балансировки нагрузки и отказоустойчивости. Решения этих вопросов сложнее, чем в распределенных СУБД, потому что должны масштабироваться на большое количество узлов. Кроме того, недавние достижения в разработке оборудования и программного обеспечения, в т. ч. сети межсоединения с низкой задержкой, многоядерные процессорные узлы, оперативная память большого объема и технология RDMA, открывают новые возможности для оптимизации. В частности, параллельные алгоритмы для самых требовательных операторов – соединения и сортировки – нужно реализовывать с учетом архитектуры NUMA.

Кластер баз данных – важный вид параллельных систем баз данных; в каждом его узле устанавливается СУБД, рассматриваемая как черный ящик. Большое число исследований посвящено тому, как воспользоваться стабильной кластерной средой для повышения производительности и доступности, применив репликацию данных. Основными их результатами стали новые методы репликации, балансировки нагрузки и обработки запросов.

8.9. Библиографические замечания

Одно из первых предложений по машинам баз данных приведено в работе [Canaday et al. 1974], где в основном рассматривается вопрос «бутылочного горлышка ввода-вывода» [Boral and DeWitt 1983], т. е. высокого времени доступа к диску по сравнению со временем доступа к оперативной памяти. Идея состояла в том, чтобы переместить функции базы данных ближе к диску. CAFS-ISP – один из первых примеров аппаратного устройства фильтрации [Babb 1979], которое встраивалось в дисковый контроллер для быстрого ассоциативного поиска. Включение универсальных микропроцессоров в дисковые контроллеры также привело к появлению интеллектуальных дисков [Keeton et al. 1998].

Первые параллельные системы баз данных – Teradata и Tandem Non-Stop-SQL – появились в начале 1980-х годов. С тех пор все основные поставщики СУБД стали поставлять параллельные версии своих продуктов. В настоящее время в этой области по-прежнему ведутся активные исследования, связанные с большими данными и применением новых возможностей оборудования, например сетей межсоединения с низкой задержкой, многоядерных процессорных узлов и оперативной памяти большого объема.

Достаточно полные обзоры литературы по параллельным системам баз данных см. в работах [DeWitt and Gray 1992, Valduriez 1993, Graefe 1993]. Параллельные архитектуры систем баз данных обсуждаются в работах [Bergsten et al. 1993, Stonebraker 1986, Pirahesh et al. 1990], их сравнение с помощью простой имитационной модели имеется в работе [Breitbart and Silberschatz 1988]. Первые архитектуры NUMA описаны в работах [Lenoski et al. 1992, Goodman and Woest 1988]. Более современный подход на основе технологии

удаленного прямого доступа к памяти (RDMA) обсуждается в работах [Novakovic et al. 2014, Leis et al. 2014, Barthels et al. 2015].

Примеры прототипов параллельных систем баз данных – системы Bubba [Boral et al. 1990], DBS3 [Bergsten et al. 1991], Gamma [DeWitt et al. 1986], Grace [Fushimi et al. 1986], Prisma/DB [Apers et al. 1992], Volcano [Graefe 1990] и XPRS [Hong 1992].

Вопрос о размещении данных, в т. ч. репликации, в параллельных системах баз данных рассматривается в работах [Livny et al. 1987, Copeland et al. 1988, Hsiao and DeWitt 1991]. Масштабируемым решением является *цепное секционирование* в системе Gamma [Hsiao and DeWitt 1991], когда основная и резервная копии хранятся в двух соседних узлах. Ассоциативный доступ к секционированному отношению с помощью глобального индекса предложен в работе [Khoshafian and Valduriez 1987].

Оптимизация параллельных запросов рассматривается в работах [Shekita et al. 1993], [Ziane et al. 1993] и [Lanzelotte et al. 1994]. Наше обсуждение модели стоимости в разделе 8.4.2.2 основано на работе [Lanzelotte et al. 1994]. Рандомизированные стратегии поиска предложены в работах [Swami 1989, Ioannidis and Wong 1987]. В системе XPRS используется двухэтапная стратегия оптимизации [Hong and Stonebraker 1993]. Оператор обмена, лежащий в основе параллельной рефрагментации при параллельной обработке запросов, был предложен в контексте системы вычисления запросов Volcano [Graefe 1990].

Существует обширная литература по параллельным алгоритмам для операторов базы данных, особенно операторов сортировки и соединения. Цель этих алгоритмов – максимизировать степень параллелизма в соответствии с законом Амдала [Amdahl 1967], утверждающим, что только часть алгоритма может быть распараллелена. В основополагающей работе [Bitton et al. 1983] предлагаются и сравниваются параллельные версии алгоритмов сортировки слиянием, соединения методом вложенных циклов и соединения методом сортировки слиянием. В работе [Valduriez and Gardarin 1984] предложено использовать хеширование в параллельных алгоритмах соединения и полусоединения. Обзор параллельных алгоритмов сортировки можно найти в работе [Bitton et al. 1984]. Описание двух основных этапов, построения и апробирования, в работе [DeWitt and Gerber 1985] было полезно для понимания параллельных алгоритмов хеш-соединения. Алгоритм хеш-соединения в системе Grace [Kitsuregawa et al. 1983], гибридный алгоритм хеш-соединения [DeWitt et al. 1984, Shatdal et al. 1994], поразрядное хеш-соединение [Manegold et al. 2002] легли в основу многих вариаций, особенно для многоядерных процессоров и архитектуры NUMA [Barthels et al. 2015]. Из других важных алгоритмов соединения отметим симметричное хеш-соединение [Wilschut and Apers 1991] и пульсирующее соединение [Haas and Hellerstein 1999b]. В работе [Barthels et al. 2015] показано, что поразрядное хеш-соединение может показывать очень хорошие результаты в крупномасштабных кластерах без разделения ресурсов с технологией RDMA.

Интерес к параллельному алгоритму соединения с помощью сортировки слиянием возродился в контексте многоядерных и NUMA-систем [Albutiu et al. 2012, Pasetto and Akhriev 2011].

Вопрос о балансировке нагрузки в параллельных системах баз данных активно изучался как в контексте систем с общей памятью и общим диском [Lu et al. 1991, Shekita et al. 1993], так и в системах без разделения ресурсов [Kitsuregawa and Ogawa 1990, Walton et al. 1991, DeWitt et al. 1992, Shatdal and Naughton 1993, Rahm and Marek 1995, Mehta and DeWitt 1995, Garofalakis and Ioannidis 1996]. Изложение модели динамической обработки в разделе 8.5 основано на работе [Bouganim et al. 1996, 1999]. Алгоритм согласования темпов описан в работе [Mehta and DeWitt 1995].

Влияние асимметричного распределения данных на параллельное выполнение изучалось в работе [Walton et al. 1991]. Общий адаптивный подход к динамическому подстраиванию степени параллелизма с использованием операторов контроля предложен в работе [Biscondi et al. 1996]. Хороший способ справиться с асимметрией данных – использовать несколько алгоритмов соединения, каждый из которых специализирован для разных степеней асимметрии, и во время выполнения определить, какой из них лучший [DeWitt et al. 1992].

Содержание раздела 8.6, посвященного отказоустойчивости, основано на работах [Kemme et al. 2001, Jiménez-Peris et al. 2002, Perez-Sorrosal et al. 2006].

Понятие кластера баз данных определено в работах [Röhm et al. 2000, 2001]. Несколько протоколов масштабируемой энергичной репликации в кластерах баз данных с использованием групповой коммуникации предложено в работах [Kemme and Alonso 2000b,a, Patiño-Martínez et al. 2000, Jiménez-Peris et al. 2002]. Их масштабируемость аналитически изучена в работе [Jiménez-Peris et al. 2003]. Частичная репликация исследовалась в работе [Sousa et al. 2001]. Изложение превентивной репликации в разделе 8.7.2 основано на работе [Pacitti et al. 2005]. Балансировка нагрузки в кластерах баз данных рассматривается в работах [Milán-Franco et al. 2004, Gançarski et al. 2007].

Большая часть раздела 8.7.4 основана на работах по адаптивному виртуальному секционированию [Lima et al. 2004] и гибридному секционированию [Furtado et al. 2008]. Физическое секционирование в кластерах баз данных, предназначенных для поддержки принятия решений, рассматривается в работе [Stöhr et al. 2000] с применением мелких секций. В работе [Akal et al. 2002] предложена классификация OLAP-запросов такая, что запросы, принадлежащие одному классу, обладают сходными свойствами распараллеливания.

Упражнения

Задача 8.1 (*). Рассмотрим кластер с общим диском и очень большие отношения, которые необходимо секционировать на несколько отдельных дисков. Как бы вы адаптировали различные методы секционирования и репликации, описанные в разделе 8.3, чтобы воспользоваться преимуществами архитектуры с общим диском? Примите во внимание влияние на производительность обработки запросов и отказоустойчивость.

Задача 8.2 (**). Хеширование с сохранением порядка [Knuth 1973] можно было бы использовать для секционирования отношения по атрибуту *A* та-

ким образом, что в кортежах из любой секции i + 1 значения A больше, чем в кортежах из секции i. Предложите параллельный алгоритм сортировки, в котором используется хеширование с сохранением порядка. Обсудите его преимущества и ограничения по сравнению с b-путевым алгоритмом сортировки слиянием из раздела 8.4.1.1.

Задача 8.3. Рассмотрим параллельный алгоритм хеш-соединения из раздела 8.4.1.2. Объясните, какие в нем этапы построения и апробирования. Верно ли, что этот алгоритм симметричен относительно входных отношений?

Задача 8.4 (*). Рассмотрим соединение двух отношений `R` и `S` в кластере без разделения ресурсов. Предположим, что `S` секционировано методом хеширования по атрибуту соединения. Модифицируйте параллельный алгоритм хеш-соединения из раздела 8.4.1.2, чтобы воспользоваться этой особенностью. Обсудите стоимость выполнения данного алгоритма.

Задача 8.5 (**). Рассмотрим простую модель стоимости для сравнения производительности трех базовых параллельных алгоритмов соединения (методом вложенных циклов, методом сортировки слиянием и методом хеширования). Она определена в терминах стоимости коммуникации (C_{COM}) и стоимости обработки (C_{PRO}). Таким образом, полная стоимость каждого алгоритма равна

$$Cost\ (Alg.) = C_{COM}(Alg.) + C_{PRO}(Alg.).$$

Для простоты в C_{COM} не включены управляющие сообщения, необходимые для запуска и завершения локальных задач. Обозначим $msg(\#tup)$ стоимость передачи сообщения, состоящего из $\#tup$ кортежей, с одного узла на другой. Стоимость обработки (включающая затраты на ввод-вывод и использование процессора) описывается функцией $C_{LOC}(m, n)$, которая вычисляет стоимость локальной обработки в случае соединения двух отношений с мощностями m и n. Предположим, что алгоритм локального соединения одинаков для всех трех параллельных алгоритмов соединения. Наконец, предположим, что объем параллельно выполняемой работы равномерно распределен между всеми узлами, выделенными оператору. Выведите формулы для вычисления полной стоимости каждого алгоритма в предположении, что входные отношения секционированы произвольным образом. При каких условиях вы рекомендовали бы использовать каждый алгоритм?

Задача 8.6. Рассмотрим следующий SQL-запрос:

```
SELECT ENAME, DUR
FROM   EMP, ASG, PROJ
WHERE  EMP.ENO=ASG.ENO
AND    ASG.PNO=PROJ.PNO
AND    RESP="Manager"
AND    PNAME="Instrumentation"
```

Для каждого из четырех возможных деревьев операторов – растущее вправо, растущее влево, зигзагообразное и кустистое – обсудите возможности распараллеливания.

Задача 8.7. Рассмотрим 9-путевое соединение (нужно соединить десять отношений). Вычислите количество возможных растущих вправо, растущих влево и кустистых деревьев в предположении, что любое отношение можно соединить с любым другим. Что вы можете сказать о параллельной оптимизации?

Задача 8.8 (**). Предложите стратегию размещения данных в NUMA-кластере (с применением технологии RDMA), которая максимизирует комбинацию *внутриузлового* параллелизма (внутриоператорного параллелизма внутри узлов с общей памятью) и *межузлового* параллелизма (межоператорного параллелизма, задействующего несколько узлов с общей памятью).

Задача 8.9 (**). Как следует изменить модель выполнения ДО, описанную в разделе 8.5.4, чтобы учесть в ней межзапросный параллелизм?

Задача 8.10 (**). Рассмотрим многопользовательскую централизованную систему баз данных. Опишите основное изменение, которое позволило бы задействовать в ней межзапросный параллелизм, с точки зрения разработчика системы и администратора. Каковы будут последствия для конечного пользователя в плане интерфейса и производительности?

Задача 8.11 (*). Рассмотрим архитектуру кластера баз данных на рис. 8.19. В предположении, что каждый узел кластера может принимать входные транзакции, детализируйте блок, представляющий кластерное ПО промежуточного уровня, описав различные уровни программного обеспечения, их компоненты и связи между ними в терминах потоков данных и управления. Какую информацию должны разделять узлы кластера? Каким образом?

Задача 8.12 (**). Обсудите проблемы отказоустойчивости протокола превентивной репликации (см. раздел 8.7.2).

Задача 8.13 (**). Сравните протокол превентивной репликации с протоколом энергичной репликации (см. главу 6) в контексте кластера базы данных со следующих точек зрения: поддерживаемые конфигурации репликации, требования к сети, согласованность, производительность, отказоустойчивость.

Задача 8.14 (**). Рассмотрим два отношения R(A,B,C,D,E) и S(A,F,G,H). Предположим, что над каждым отношением существует кластерный индекс по атрибуту. В предположении, что имеется кластер баз данных с полной репликацией, для каждого из следующих запросов определите, можно ли воспользоваться виртуальным секционированием для достижения внутризапросного параллелизма, и, если да, напишите соответствующий подзапрос и окончательный композиционный запрос.

a)
```
SELECT B, COUNT(C)
FROM   R
GROUP  BYB
```

b)
```
SELECT C, SUM(D), AVG(E)
FROM R
WHERE B=:v1
GROUP BY C
```

c)
```
SELECT B, SUM(E)
FROM   R, S
WHERE  R.A=S.A
GROUP BY B
HAVING COUNT(*) > 50
```

d)
```
SELECT B, MAX(D)
FROM   R, S
WHERE C = (SELECT SUM(G) FROM S WHERE S.A=R.A)
GROUP BY B
```

e)
```
SELECT B, MIN(E)
FROM R
WHERE D > (SELECT MAX(H) FROM S WHERE G >= :v1)
GROUP BY B
```

Глава 9

Управление данными в одноранговых системах

В этой главе мы обсудим проблемы управления данными в «современных» одноранговых системах (P2P). Мы намеренно употребили слово «современные», чтобы отличить эти системы от ранних P2P-систем, существовавших до появления клиент-серверных технологий. Как было сказано в главе 1, первые работы по распределенным СУБД были в основном ориентированы на P2P-архитектуры, в которых функциональность всех узлов системы была одинакова. Так что в каком-то смысле одноранговое управление данными – старая технология, если понимать под этим отсутствие выделенных «клиентов» и «серверов». Однако «современные» P2P-системы переросли такую простую характеристику и отличаются от старых систем, носящих такое же название, во многих важных отношениях.

Первое отличие – значительная распределенность. Если в ранних системах было немного узлов (от силы несколько десятков), то в современных число узлов исчисляется тысячами. Кроме того, узлы сильно разнесены территориально, причем в нескольких пунктах могут формироваться кластеры.

Второе отличие – внутренне присущая узлам гетерогенность и автономность. Вкупе с сильной распределенностью это исключает из рассмотрения некоторые подходы, применяемые в распределенных базах данных.

Третье важное отличие – нестабильность таких систем. Распределенные СУБД работают в строго контролируемой среде, где добавление новых и удаление существующих узлов производят редко и очень аккуратно. В современных P2P-системах узлами очень часто являются персональные компьютеры, которые присоединяются к системе и покидают ее по воле хозяев, создавая значительные трудности для управления данными.

В этой главе мы будем заниматься современным воплощением P2P-систем. К ним предъявляются следующие требования:

- **автономность.** Автономный узел должен иметь возможность присоединиться к системе или покинуть ее в любой момент времени без каких-либо ограничений. Он также должен иметь возможность контролировать данные, которые хранит сам, и определять, какие еще узлы могут хранить его данные (например, некоторые узлы могут быть доверенными);

- **выразительная способность запросов.** Язык запросов должен позволять пользователю описывать нужные ему данные с подходящим уровнем детализации. В простейшей форме запрос содержит только ключ, что годится лишь для поиска файлов. Поиск по ключевым словам с ранжированием результатов подходит для поиска документов, но для более структурированных данных нужен язык запросов типа SQL;
- **эффективность.** Использование ресурсов P2P-системы (пропускная способность, вычислительные мощности, запоминающие устройства) должно стоить дешево, а количество запросов, обработанных системой в течение заданного времени, должно быть велико;
- **качество обслуживания.** Под этим понимается воспринимаемая пользователем эффективность системы, например полнота результатов, согласованность данных, доступность данных и время ответа на запрос;
- **отказоустойчивость.** Эффективность и качество обслуживания должны быть гарантированы, несмотря на отказы узлов. Принимая во внимание динамическую природу одноранговых узлов, которые могут появляться и исчезать в любое время, важно правильно использовать репликацию данных;
- **безопасность.** Открытость P2P-системы влечет за собой серьезные проблемы в плане безопасности, потому что ни одному серверу нельзя доверять. В том, что касается управления данными, главная проблема – контроль доступа и, в частности, защита прав интеллектуальной собственности на контент.

Разработано много P2P-систем для различных применений: совместные вычисления (например, проект SETI@home), связь (например, ICQ), обмен данными (например, BitTorrent, Gnutella и Kazaa). Естественно, нам интересны системы обмена данными. Такие популярные системы, как BitTorrent, Gnutella и Kazaa, весьма ограничены с точки зрения функциональности базы данных. Во-первых, они предлагают разделение данных только на уровне файлов без сколько-нибудь развитых средств поиска и запросов по содержимому. Во-вторых, это приложения, ориентированные на решение одной-единственной задачи, и не вполне понятно, как обобщить их на другие функции. В этой главе мы обсудим исследования в направлении предоставления настоящей функциональности баз данных поверх инфраструктуры P2P. Предстоит решить следующие проблемы:

- местоположение данных: одноранговые узлы должны иметь возможность ссылаться на данные, хранящиеся в других узлах, и находить их;
- обработка запросов: получив запрос, система должна уметь находить узлы, где хранятся релевантные данные, и эффективно выполнять запрос;
- интеграция данных: даже если источники разделяемых данных в системе пользуются разными схемами или представлениями, узлы все равно должны иметь возможность обратиться к данным, в идеале пользуясь тем представлением, с помощью которого моделируют собственные данные;
- согласованность данных: если данные реплицируются или кешируются, то ключевой вопрос – как поддержать согласованность копий.

На рис. 9.1 показана эталонная архитектура одноранговых узла, участвующего в P2P-системе разделения данных. В зависимости от функциональности системы некоторые компоненты могут отсутствовать, могут быть объединены в один или реализованы с помощью специализированных узлов. Ключевой момент предложенной архитектуры – разделение функциональности на три главных компонента: (1) интерфейс для отправки запросов; (2) уровень управления данными, который отвечает за обработку запросов и метаданные (например, службу каталогов); (3) инфраструктура P2P, которая состоит из подуровня P2P-сети и самой P2P-сети. В этой главе мы будем рассматривать уровень управления данными P2P и инфраструктуру P2P.

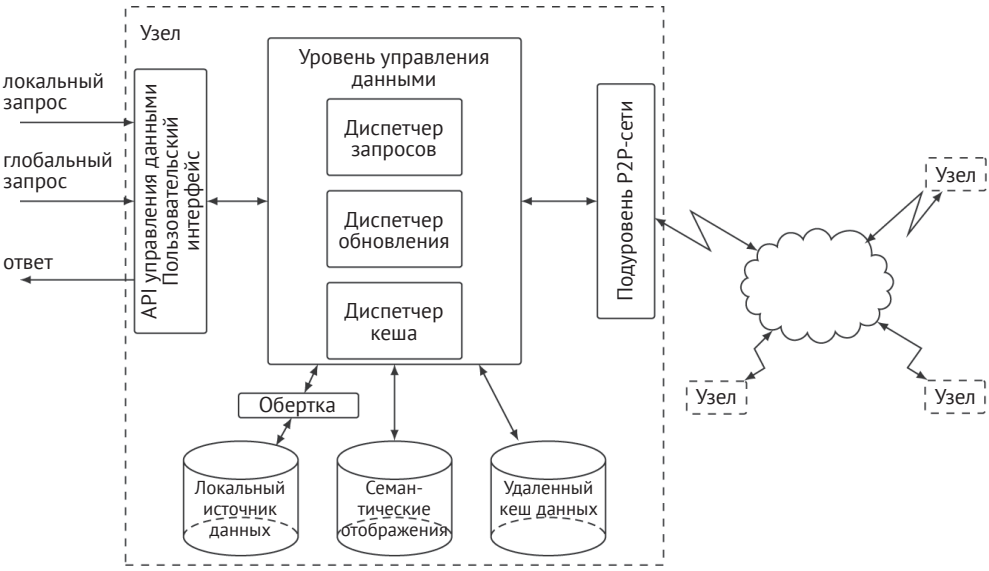


Рис. 9.1 ❖ Эталонная архитектура одноранговых узла

Запросы отправляются с помощью пользовательского интерфейса или API управления данными и обрабатываются уровнем управления данными. Они могут ссылаться на данные, хранящиеся локально или где-то в другом месте системы. Запрос обрабатывается модулем диспетчера запросов, который извлекает информацию о семантическом отображении из репозитория, если в систему интегрированы гетерогенные источники данных. Репозиторий семантических отображений содержит метаданные, которые позволяют диспетчеру запросов находить в системе узлы, содержащие данные, релевантные запросу, и переформулировать исходный запрос, так чтобы его понимали другие узлы. В некоторых P2P-системах семантические отображения могут храниться в специализированных узлах. В таком случае диспетчер запросов должен будет обратиться к этим узлам или передать им запрос для выполнения. Если все источники данных в системе пользуются одной и той же схемой, то ни репозиторий семантических отображений, ни функции, относящиеся к переформулированию запроса, не нужны.

В предположении, что репозиторий семантических отображений существует, диспетчер запросов вызывает службы, реализованные подуровнем P2P-сети, для взаимодействия с другими узлами, участвующими в выполнении запроса. На способ выполнения запроса влияет реализация инфраструктуры P2P. В одних системах данные отправляются узлу в момент инициирования запроса, а затем объединяются в этом узле. В других системах имеются специализированные узлы для выполнения запросов и координации. В любом случае результаты, возвращенные узлами, участвующими в выполнении запроса, могут кешироваться локально для ускорения выполнения подобных запросов в будущем. Диспетчер кеша поддерживает локальный кеш в каждом узле. Но бывает и так, что кеширование производится только в специализированных узлах.

Диспетчер запросов отвечает также за выполнение локальной части глобального запроса, когда удаленный узел запрашивает данные. Обертка может скрывать данные, язык запросов и прочие несовместимости между локальным источником данных и уровнем управления данными. Когда данные обновляются, диспетчер обновления координирует выполнение обновления всеми узлами, в которых хранятся реплики обновляемых данных. Инфраструктура P2P-сети, которая может иметь структурированную или неструктурированную топологию, предоставляет службы связи уровню управления данными.

Далее в данной главе мы рассмотрим все компоненты этой эталонной архитектуры, начав с вопроса об инфраструктуре (раздел 9.1). Проблемы отображения данных и подходы к их решению обсуждаются в разделе 9.2. Обработка запросов – тема раздела 9.3. Вопросы согласованности данных и репликации обсуждаются в разделе 9.4. В разделе 9.5 мы познакомимся с блокчейном – P2P-инфраструктурой для эффективного, безопасного и постоянного сохранения транзакций.

9.1. Инфраструктура

В основе инфраструктуры любой P2P-системы лежит P2P-сеть, построенная поверх физической сети (обычно интернета), поэтому ее обычно называют *наложенной* (или *оверлейной*) *сетью*. Топологии наложенной и физической сетей могут отличаться (и обычно отличаются), а все алгоритмы направлены на оптимизацию взаимодействия в наложенной сети (обычно в терминах количества пересылок сообщения на пути от исходного узла к конечному – оба в наложенной сети). Различие между физической и наложенной сетями может стать проблемой, поскольку два узла, являющихся соседями в наложенной сети, физически могут отстоять друг от друга на большое расстояние. Поэтому стоимость коммуникации в наложенной сети может не отражать истинных затрат на коммуникацию в сети физической. В должное время мы обсудим эту проблему.

Наложенные сети бывают двух видов: чистые и гибридные. В ч*истой наложенной сети* (чаще встречается выражение *чистая P2P-сеть*) между узлами

нет никаких различий– все узлы одинаковы. С другой стороны, в *гибридной P2P-сети* некоторым узлам поручены специальные задачи. Гибридные сети часто называют *супероднорангoвыми системами*, поскольку некоторые узлы «контролируют» другие узлы в подведомственной им части. Чистые сети подразделяются на структурированные и неструктурированные. В *структурированной сети* реализован строгий контроль над топологией и маршрутизацией сообщений, а в *неструктурированной* каждый узел может напрямую взаимодействовать со своими соседями и включиться в сеть, прикрепившись к любому узлу.

9.1.1. Неструктурированные P2P-сети

В неструктурированных P2P-сетях нет никаких ограничений на размещение данных в наложенной сети. Наложенная сеть создается недетерминированным образом, а размещение данных никак не связано с топологией наложения. Каждый узел знает своих соседей, но не знает о ресурсах, которыми они владеют. На рис. 9.2 приведен пример неструктурированной P2P-сети.

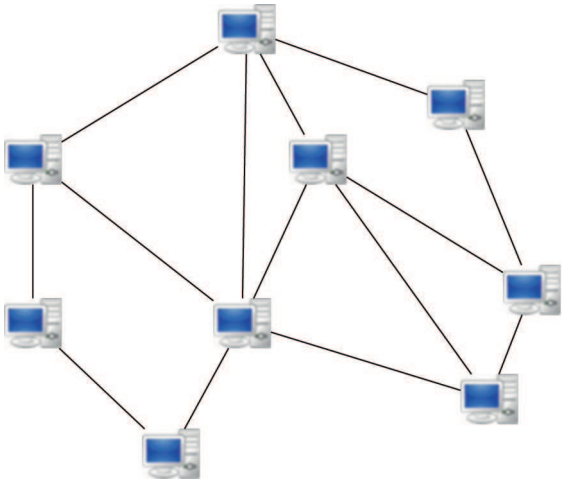


Рис. 9.2 ❖ Неструктурированная P2P-сеть

Неструктурированные сети – самые ранние образцы P2P-систем, функциональность которых сводилась к обмену файлами. В узлах этих систем хранились реплицированные копии популярных файлов, так что не было нужды загружать их с централизованного сервера. Примерами таких систем могут служить Gnutella, Freenet, Kazaa и BitTorrent.

Фундаментальный вопрос всех P2P-сетей – тип индекса ресурсов, которыми владеет каждый узел, поскольку от него зависит способ поиска ресурсов. То, что в контексте P2P-систем называется «управлением индексом», очень похоже на управление каталогом, которое мы изучали в главе 2. В индексах

хранятся метаданные, поддерживаемые системой. Точный состав метаданных зависит от системы, но как минимум в него входит информация о ресурсах и размерах.

Существует два способа поддержания индексов: централизованный, когда в одном узле хранятся метаданные для всей P2P-системы, и распределенный, когда в каждом узле хранятся метаданные о ресурсах, которыми он владеет. Как видим, варианты те же, что для управления каталогом.

От типа индекса в P2P-системе (централизованный или распределенный) зависит, как производится поиск ресурсов. Заметим, что сейчас мы не говорим о выполнении запросов, а просто обсуждаем, как, зная идентификатор ресурса, инфраструктура P2P может найти сам ресурс. В системах с централизованным индексом необходимо запросить у центрального узла местонахождение ресурса, а затем обратиться напрямую к самому узлу, где этот ресурс хранится (рис. 9.3). Таким образом, система работает как клиент-серверная до момента получения информации из индекса (т. е. метаданных), но затем взаимодействуют только два одноранговых узла. Отметим, что центральный узел может вернуть множество узлов, владеющих ресурсом, и запрашивающий узел должен выбрать один из них. Но, возможно, центральный узел сам производит выбор (например, принимая во внимание нагрузку и условия в сети) и возвращает только один рекомендуемый узел.

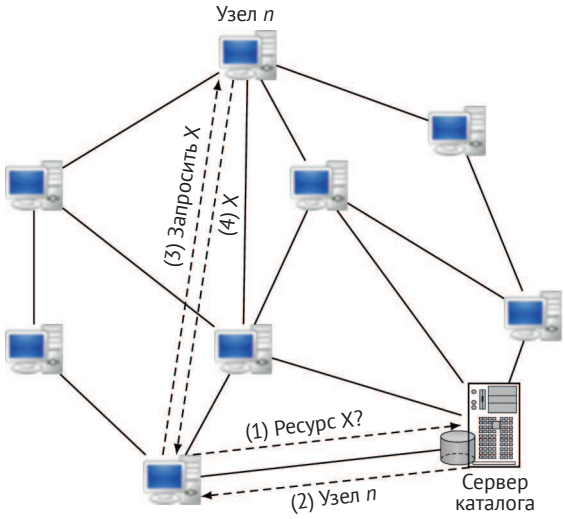


Рис. 9.3 ❖ Поиск в централизованном индексе:
(1) узел запрашивает местонахождение ресурса у диспетчера центрального индекса;
(2) в ответе указан идентификатор узла, владеющего ресурсом;
(3) узлу посылается запрос на ресурс; (4) ресурс передается

В системах с распределенным индексом существует несколько вариантов поиска. Самый популярный – лавинный запрос, когда узел, которому нужен ресурс, посылает запрос всем своим соседям в наложенной сети. Если у какого-то соседа имеется ресурс, то он отвечает, в противном случае каж-

дый сосед переправляет запрос своим соседям, пока ресурс не будет найден или в наложенной сети не останется неопрошенных узлов (рис. 9.4).

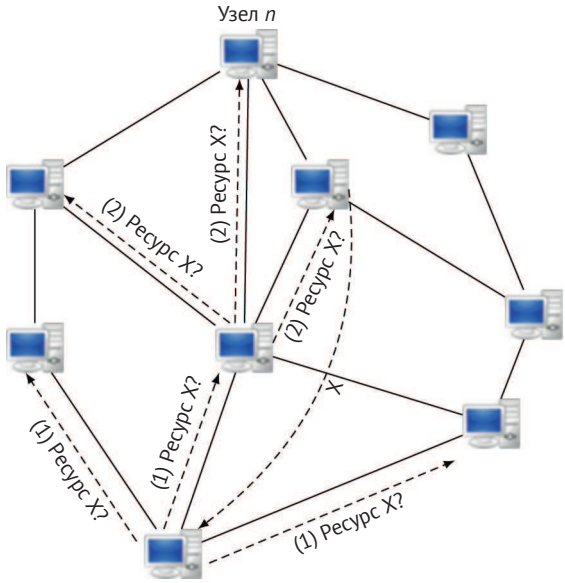


Рис. 9.4 ❖ Поиск в децентрализованном индексе:
(1) узел запрашивает местонахождение ресурса у всех своих соседей;
(2) каждый сосед переправляет запрос своим соседям, если не владеет ресурсом;
(3) узел, владеющий ресурсом, отвечает на запрос, посылая этот ресурс

Естественно, лавинный запрос сильно нагружает сеть и не масштабируется – по мере увеличения наложенной сети объем передаваемой информации растет. Эта проблема решается ограничением времени жизни (Time-to-Live – TTL), т. е. максимального количества пересылок сообщения: как только оно обращается в ноль, сообщение удаляется из сети. Однако TTL также ограничивает количество достижимых узлов.

Для решения проблемы были предложены и другие подходы. Самый прямолинейный метод – когда каждый узел выбирает подмножество своих соседей и адресует запрос только им. Есть разные способы выбрать это подмножество. Например, можно воспользоваться идеей случайного блуждания, когда каждый узел выбирает соседа случайным образом и направляет запрос только ему. Вместо этого каждый сосед может хранить не только индекс локальных ресурсов, но также информацию о ресурсах, которыми владеют узлы в некотором радиусе от него самого, а при маршрутизации запросов использовать историческую информацию об их производительности. Еще один вариант – использовать похожие индексы, основанные на сведениях о ресурсах в каждом узле, чтобы вернуть список соседей, которые с наибольшей вероятностью находятся в направлении узла, владеющего запрошенными ресурсами. Такие индексы называются маршрутными и чаще применяются в структурированных сетях, так что более подробно мы обсудим их позже.

При другом подходе используется *протокол сплетен*, или *протокол распространения эпидемии*. Первоначально он был предложен для поддержания взаимной согласованности реплицированных данных путем распространения обновлений реплик на все узлы сети. Но впоследствии успешно применялся в P2P-сетях для распространения данных. В основе своей протокол сплетен прост. У каждого узла имеется полное представление о сети (список адресов всех узлов), и он случайным образом выбирает какой-нибудь узел для распространения запроса. Основное преимущество протокола сплетен – устойчивость к отказам узлов, потому что с очень высокой вероятностью запрос в конечном итоге попадает всем узлам сети. Но в крупных P2P-сетях простая модель распространения сплетен не масштабируется, потому что поддержание полного представления о сети в каждом узле требует очень высокого трафика. Решение проблемы заключается в том, чтобы хранить в каждом узле лишь частичное представление, например список из нескольких десятков соседних узлов. Для распространения сплетни запрос посылается узлу, случайным образом выбранному из частичного представления. Кроме того, узлы, участвующие в распространении сплетни, обмениваются своими частичными представлениями, чтобы отразить изменения в сети. Таким образом, постоянно обновляя свои частичные представления, узлы самоорганизуются в рандомизированную наложенную сеть, которая очень хорошо масштабируется.

И последнее, что мы хотели бы обсудить в связи с неструктурированными сетями, – вопрос о том, как узлы присоединяются к сети и покидают ее. Процедура зависит от того, является ли индекс централизованным или распределенным. В системе с централизованным индексом узел, желающий присоединиться к сети, просто связывается с узлом, владеющим индексом, и информирует его о том, какие ресурсы он готов включить в P2P-систему. В случае же распределенного индекса входящий узел должен знать о каком-нибудь узле системы, к которому он «прикрепляется» и от которого получает информацию о его соседях. После завершения этой процедуры узел становится частью системы и начинает строить список своих соседей. Узлам, покидающим систему, не нужно ничего делать, они просто исчезают. Со временем их отсутствие будет обнаружено, и наложенная сеть скорректируется.

9.1.2. Структурированные P2P-сети

Структурированные P2P-сети призваны решить проблемы масштабируемости, с которыми сталкиваются неструктурированные сети. Для этого в них строго контролируется топология наложенной сети и размещение ресурсов. Поэтому они достигают более высокой масштабируемости ценой меньшей автономности, поскольку каждый узел, присоединяющийся к сети, разрешает размещать свои ресурсы там, где сочтет нужным конкретный метод управления.

Как и в неструктурированных P2P-сетях, необходимо решить два фундаментальных вопроса: как индексируются ресурсы и как производится их поиск. Самый популярный метод индексирования и поиска ресурсов в структу-

рированных P2P-сетях называется *распределенной хеш-таблицей* (distributed hash table – DHT). Системы на основе DHT предоставляют две функции API: `put(key, data)` и `get(key)`, где `key` – идентификатор объекта. Каждый ключ (k_i) хешируется для получения идентификатора узла (p_i), в котором будут храниться данные объекта (рис. 9.5).

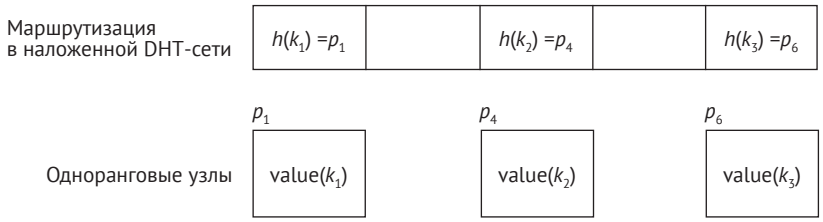


Рис. 9.5 ❖ DHT-сеть

Простейший подход – использовать URI или IP-адрес в качестве идентификатора узла-хранителя ресурса. Однако одно из важных проектных требований состоит в том, что ресурсы должны быть распределены в наложенной сети равномерно, а ни URI, ни IP-адреса не обладают в этом отношении достаточной гибкостью. Поэтому применяются методы *согласованного хеширования*, которые обеспечивают равномерное хеширование значений. Хотя для генерирования *отображений виртуальных адресов* можно воспользоваться различными хеш-функциями, чаще всего в качестве базовой[1] хеш-функции выбирают SHA-1, поскольку этот алгоритм обеспечивает равномерность распределения и безопасность (гарантируя целостность ключей). Детали проектирования хеш-функции зависят от реализации, и мы не будем углубляться в этот вопрос.

Поиск в структурированной P2P-сети на основе DHT также включает хеш-функцию: ключ ресурса хешируется, чтобы получить идентификатор узла в наложенной сети, отвечающего за этот ключ. Затем в наложенной сети инициируется поиск для нахождения этого узла. Этот *протокол маршрутизации* зависит от реализации и тесно связан со структурой используемой наложенной сети. Ниже мы обсудим один пример.

Все протоколы маршрутизации стремятся обеспечить эффективность поиска, но, кроме того, их цель – минимизировать объем *маршрутной информации* (или *состояния маршрутизации*), хранящейся в маршрутной таблице в каждом узле наложенной сети. Конкретный состав этой информации зависит от протокола маршрутизации и структуры наложенной сети, но ее должно быть достаточно, чтобы запросы `put` и `get` могли быть доставлены узлу-адресату. Любая реализация таблицы маршрутизации нуждается в алгоритме обслуживания, который поддерживает маршрутизацию в актуальном и согласованном состоянии. В отличие от маршрутизаторов в интернете, которые также поддерживают базы данных о маршрутизации, P2P-системы ставят

[1] Базовой называется хеш-функция, которая используется в качестве основы для проектирования другой хеш-функции.

более сложные проблемы, поскольку множество узлов крайне нестабильно, а каналы связи ненадежны. Кроме того, DHT должна обладать идеальной полнотой (т. е. все ресурсы, доступные по данному ключу, должны быть найдены), поэтому обеспечение согласованности состояния маршрутизации представляет собой важнейшую задачу. Таким образом, поддержание согласованного состояния маршрутизации в условиях конкурентных запросов поиска и в периоды высокой изменчивости сети – принципиальное требование.

Было предложено много наложенных сетей на основе DHT. Их можно классифицировать по *геометрии маршрутизации* и *алгоритму маршрутизации*. Геометрия маршрутизации определяет, как организованы соседи и маршруты. Алгоритм маршрутизации соответствует вышеупомянутому протоколу маршрутизации и определяет, как выбирается следующий переход (маршрут) при заданной геометрии маршрутизации. Перечислим наиболее распространенные из существующих наложенных сетей на основе DHT.

- **Дерево.** Листовые узлы соответствуют идентификаторам узлов, в которых хранятся ресурсы с искомыми ключами. Высота дерева равна $\log n$, где n – количество узлов в нем. Процесс поиска начинается с корня и продвигается в направлении листьев, на каждом шаге выбирается узел с наибольшим совпадающим префиксом. Таким образом, совпадение можно интерпретировать как исправление битов слева направо при каждом переходе в дереве. Популярная реализация DHT, относящаяся к этой категории, – Tapestry, в ней используется *суррогатная маршрутизация*, чтобы переправлять запросы в каждом узле к ближайшему числу в таблице маршрутизации. Суррогатная маршрутизация определяется как маршрутизация к *ближайшему* числу, если не удается найти совпадение с самым длинным префиксом. В Tapestry с каждым уникальным идентификатором ассоциируется узел, являющийся корнем уникального остовного дерева, используемого для маршрутизации сообщений с данным идентификатором. Таким образом, поиск производится от нижнего уровня остовного дерева вверх к корневому узлу идентификатора. Хотя геометрия маршрутизации в Tapestry несколько отличается от традиционных древовидных структур, она тесно связана со структурой дерева, поэтому мы отнесли ее к этому классу.
  В древовидных структурах соседа каждого узла можно выбрать из 2^{i-1} узлов в поддереве, с которыми он имеет $\log(n - i)$ общих префиксных бит. Количество потенциальных соседей экспоненциально возрастает по мере продвижения вверх по дереву. Таким образом, всего имеется $n^{\log n/2}$ возможных таблиц маршрутизации на один узел (отметим, однако, что для каждого узла можно выбрать только одну такую таблицу). Поэтому геометрия дерева обладает хорошими характеристиками выбора соседей, которые обеспечивают отказоустойчивость. Однако при отправке сообщения конкретному узлу для маршрутизации можно использовать только один соседний узел, поэтому DHT не дает никакой гибкости при выборе маршрута.
- **Гиперкуб.** Гиперкубическая геометрия маршрутизации основана на d-мерном декартовом пространстве, разбитом на множество зон, так что каждый узел отвечает за одну зону. Примером гиперкубической

DHT является сеть, адресуемая по содержимому (Content Addressable Network – CAN). В d-мерном пространстве каждый узел имеет $2d$ соседей (в этом обсуждении будем считать, что $d = \log n$). Если считать, что каждая координата представляет множество битов, то идентификатор каждого узла можно представить битовой строкой длины $\log n$. В этом отношении гиперкубическая геометрия очень похожа на дерево, потому что в ней также *исправляются* биты при каждом переходе на пути к месту назначения. Однако в гиперкубе соседние узлы отличаются *ровно* одним битом, поэтому каждый переадресующий узел должен изменить только один бит в битовой строке, что можно делать в любом порядке. Стало быть, первая коррекция может быть применена к любому из $\log n$ бит, следующая коррекция – к любому из $\log n - 1$ бит и т. д. Следовательно, всего имеется $(\log n)!$ возможных маршрутов между узлами, так что гиперкубическая геометрия маршрутизации обладает высокой гибкостью. Однако узел в пространстве не может выбирать координаты соседей, потому что соседние зоны жестко фиксированы. Поэтому в гиперкубе выбор соседей недостаточно гибкий.

- **Кольцо.** Кольцевая геометрия представлена одномерным круговым пространством идентификаторов, в котором узлы размещены в разных точках окружности. Расстоянием между любыми двумя узлами на окружности считается разность числовых значений идентификаторов (по часовой стрелке). Поскольку окружность одномерна, идентификаторы данных можно представлять десятичными числами (записанными в виде битовых строк), которые отображаются на узел, ближайший к заданному десятичному числу в пространстве идентификаторов. Популярным примером кольцевой геометрии является система Chord. В Chord узел с идентификатором a хранит информацию о $\log n$ соседях по кольцу, где i-м соседом считается узел, ближайший к $a + 2^{i-1}$ на окружности. Пользуясь этими связями (которые называются *указателями*, англ. finger), Chord может найти маршрут к любому узлу, состоящий не более чем из $\log n$ звеньев.

  Скрупулезный анализ структуры Chord показывает, что в узле необязательно хранить узел, ближайший к $a + 2^{i-1}$, в качестве соседа. На самом деле верхнюю границу поиска – $\log n$ – можно соблюсти, даже если выбирать любой узел из диапазона $[(a + 2^{i-1}), (a + 2^i)]$. Так что если говорить о гибкости маршрутизации, то каждый узел может выбрать одну из $n^{\log n/2}$ таблиц маршрутизации. Это обеспечивает большую гибкость в выборе соседа. Кроме того, при построении маршрута к любому узлу первый переход можно выбирать между $\log n$ соседями, следующий между $\log n - 1$ соседями и т. д. Поэтому всего имеется $(\log n)!$ возможных маршрутов к месту назначения, и, следовательно, кольцевая геометрия также обладает высокой гибкостью в выборе маршрута.

  Эти геометрии самые популярные, но есть много других основанных на DHT структурированных наложенных сетей с различными топологиями.

  Основанные на DHT наложенные сети эффективны в том смысле, что

гарантируют нахождение узла с нужными данными за $\log n$ переходов, где n – количество узлов в системе. Однако у них есть несколько проблем, в частности при рассмотрении их с точки зрения управления данными. Поскольку в DHT для лучшего распределения ресурсов используются функции согласованного хеширования, два узла, являющихся «соседями» в наложенной сети в силу близости хеш-значений, могут далеко отстоять друг от друга в физической сети. Поэтому при обмене данными с соседом по наложенной сети может возникать большая задержка. Для преодоления этой трудности были проведены исследования по проектированию *локально-чувствительных* хеш-функций. Еще одна трудность заключается в отсутствии какой-либо гибкости при размещении данных – элемент данных надлежит поместить в узел, определенный хеш-функцией. Поэтому если существуют одноранговые узлы, желающие привнести собственные данные, то они должны дать согласие на перемещение своих данных в другие узлы. Это создает проблемы с точки зрения автономности узлов. Наконец, в сетях на основе DHT, как и в любых структурах с хеш-индексами, трудно выполнить запрос по диапазону. Исследования с целью решить эту проблему мы обсудим ниже.

Эти сложности стали стимулом для разработки структурированных наложенных сетей, в которых для маршрутизации не используется DHT. В этих системах узлы отображаются в пространство данных, а не в пространство хеш-ключей. Существует несколько способов разделить пространство данных между несколькими одноранговыми узлами.

- **Иерархическая структура.** Во многих системах используются наложенные сети с иерархической структурой: деревья, сбалансированные деревья, рандомизированные сбалансированные деревья (например, списки с пропусками) и другие. Так, в системах PHT и P-Grid применяется структура двоичного префиксного дерева, в котором узлы, хранящие данные с общими префиксами, сосредоточены в общих ветвях. Сбалансированные деревья также широко применяются, т. к. гарантируют высокую эффективность маршрутизации (ожидаемое количество переходов между любыми двумя узлами пропорционально высоте дерева). Например, в системах BATON, VBI-tree и BATON* для управления узлами используется k-путевое сбалансированное дерево, и данные равномерно распределены между узлами на уровне листьев. Для сравнения в системе P-Tree используется структура B-дерева, более гибкая с точки зрения структурных изменений. Системы SkipNet и Skip Graph основаны на списке с пропусками, узлы в них связаны в рандомизированное сбалансированное дерево, в котором порядок узла определяется хранимыми в нем данными.
- **Кривая, заполняющая пространство.** Эта архитектура обычно применяется для линеаризации отсортированных данных в многомерном пространстве. Узлы располагаются на кривой, заполняющей пространство (например, кривой Гильберта), так что возможен обход узлов в порядке данных.
- **Структура гиперпрямоугольника.** В таких системах каждое измере-

ние гиперпрямоугольника соответствует одному атрибуту данных в соответствии с тем, какая желательна организация. Узлы распределены в пространстве данных либо равномерно, либо с учетом локализации данных (например, с помощью связи, определяемой пересечением данных). Гиперпрямоугольное пространство затем распределяется между узлами на основе их геометрического положения, а соседние узлы связываются между собой и образуют наложенную сеть.

9.1.3. Супероднoранговые P2P-сети

Супероднoранговые P2P-системы представляют собой гибрид чистых P2P-систем и традиционных клиент-серверных архитектур. Они похожи на клиент-серверные архитектуры тем, что не все узлы одинаковы; некоторые узлы (они называются *суперузлами*) играют роль выделенных серверов для остальных узлов и могут выполнять такие сложные функции, как индексирование, обработка запросов, контроль доступа и управление метаданными. Если в системе существует только один суперузел, то мы получаем клиент-серверную архитектуру. Однако они считаются P2P-системами, потому что суперузлы организованы, как в P2P-сети, и могут взаимодействовать между собой нетривиальными способами. Таким образом, в отличие от клиент-серверных систем, глобальная информация необязательно централизована, она может быть секционирована между суперузлами или реплицирована в них.

В супероднoранговой сети запрашивающий узел посылает запрос, возможно выраженный на языке высокого уровня, своему суперузлу. Суперузел может найти релевантные узлы либо напрямую с помощью своего индекса, либо косвенно, воспользовавшись услугами соседних суперузлов. Точнее, поиск ресурса производится следующим образом (см. рис. 9.6):

1) узел, скажем узел 1, запрашивает ресурс, посылая запрос своему суперузлу;
2) если ресурс находится в одном из узлов, контролируемых этим суперузлом, то он уведомляет узел 1, после чего два узла начинают взаимодействовать между собой;
3) если ни в одном из узлов, контролируемых суперузлом, искомого ресурса нет, то суперузел запрашивает другие суперузлы. Суперузел того узла, который содержит ресурс (скажем, узла *n*), отвечает запросившему суперузлу;
4) идентификатор узла *n* посылается узлу 1, после чего оба узла могут взаимодействовать напрямую с целью получения ресурса.

Основные преимущества супероднoранговых сетей – эффективность и качество обслуживания (например, полнота результатов, время ответа на запрос). Время, необходимое для поиска данных путем прямого обращения к индексам в суперузле, очень мало по сравнению с лавинным запросом. Кроме того, в супероднoранговых сетях задействуются различные возможности узлов в части мощности процессоров, пропускной способности и емкости запоминающих устройств, поскольку суперузлы берут на себя значительную часть сетевой нагрузки. Контроль доступа также упрощается, потому что ин-

формацию о каталоге и о безопасности можно хранить в суперузлах. Однако автономность при этом ограничена, т. к. узлы не могут беспрепятственно обращаться к любому суперузлу. Отказоустойчивость обычно ниже, потому что суперузлы являются точками общего отказа для подведомственных им узлов (динамическая замена суперузла может сгладить эту проблему).

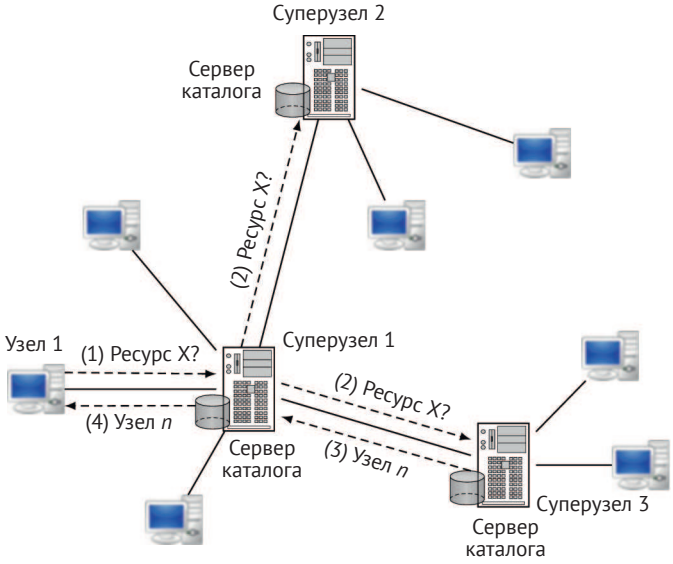


Рис. 9.6 ❖ Поиск в супероднорангoвой системе:
(1) узел запрашивает местонахождение ресурса у своего суперузла;
(2) при необходимости суперузел посылает запрос другим суперузлам;
(3) суперузел, один из подведомственных узлов которого содержит искомый ресурс, отвечает, сообщая адрес этого узла; (4) суперузел уведомляет исходный запросивший узел

Примерами супероднорангoвых сетей могут служить Edutella и JXTA.

9.1.4. Сравнение P2P-сетей

На рис. 9.7 показано, как требования к управлению данными (автономность, выразительная способность запроса, эффективность, качество обслуживания, отказоустойчивость, безопасность) удовлетворяются в P2P-сетях трех основных классов. Это лишь приблизительное сравнение, позволяющее составить представление о сравнительных достоинствах каждого класса. Очевидно, что в любом классе есть простор для совершенствования. Например, отказоустойчивость супероднорангoвых систем можно повысить, воспользовавшись репликацией и отработкой отказа. Выразительную способность запроса можно улучшить, поддержав более сложные запросы поверх структурированной сети.

Требования	Неструктурированная	Структурированная	Супероднoранговая
Автономность	Низкая	Низкая	Умеренная
Выразительная способность запроса	Высокая	Низкая	Высокая
Эффективность	Низкая	Высокая	Высокая
Качество обслуживания	Низкое	Высокое	Высокое
Отказоустойчивость	Высокая	Высокая	Низкая
Безопасность	Низкая	Низкая	Высокая

Рис. 9.7 ❖ Сравнение различных подходов

9.2. Отображение схем в P2P-системах

В главе 7 мы обсуждали важность и методы проектирования систем интеграции баз данных. Аналогичные вопросы возникают и при разделении данных в P2P-системах.

В силу специфики P2P-систем, в частности динамичности и автономности узлов, подходы, опирающиеся на централизованные глобальные схемы, неприменимы. Основная проблема – как поддержать децентрализованное отображение схем, чтобы запрос, выраженный в терминах схемы одного узла, можно было переформулировать и адресовать к схеме другого узла. Применяемые в P2P-системах подходы к определению и созданию отображений между схемами узлов можно классифицировать следующим образом: попарное отображение схем, отображение, основанное на методах машинного обучения, отображение на основе общего согласия и отображение схем методами информационного поиска.

9.2.1. Попарное отображение схем

В этом случае каждый пользователь определяет отображение между локальной схемой и схемами всех узлов, содержащих интересующие его данные. Опираясь на транзитивность определенных отображений, система пытается построить отображения между схемами, для которых отображение не определено.

Этот подход принят в системе Piazza (рис. 9.8). Разделяемые данные представлены XML-документами, и в каждом узле имеется схема, которая определяет местную терминологию и структурные ограничения. Когда новый узел в первый раз присоединяется к системе, он отображает свою схему на схемы каких-то других узлов. Определение каждого отображения начинается с XML-шаблона, который сопоставляется с некоторым путем или поддеревом целевой схемы. Элементы шаблона можно аннотировать выражениями запросов, которые связывают переменные с XML-узлами исходной схемы.

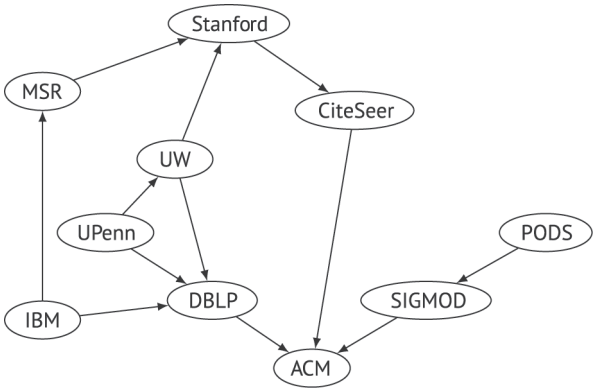


Рис. 9.8 ❖ Пример попарного отображения схем в системе Piazza

Локальная реляционная модель (Local Relational Model – LRM) – другой пример этого подхода. В LRM предполагается, что в узлах находятся реляционные базы данных и каждый узел знает, с какими узлами он может обмениваться данными и службами. Это множество узлов называется *знакомыми* данного узла. Каждый узел должен определить семантические зависимости и правила трансляции между своими данными и данными каждого знакомого. Определенные отображения образуют семантическую сеть, которая используется для переформулирования запросов в P2P-системе.

В системе Hyperion этот подход обобщен таким образом, что автономные узлы могут формировать знакомства во время выполнения, используя таблицы отображений для определения соответствия значений между гетерогенными базами данных. Узлы выполняют локальную обработку запросов и обновлений, а также распространяют запросы и обновления своим знакомым узлам.

В системе P-Grid также предполагается существование попарных отображений между узлами, которое первоначально строится опытными экспертами. Опираясь на транзитивность этих отображений и применяя протокол сплетен, P-Grid выводит новые отображения, связывающие схемы узлов, между которыми изначально не было определено отображение.

9.2.2. Отображение на основе методов машинного обучения

Этот подход обычно используется, когда разделяемые данные определены на основе онтологий и таксономий, как в семантическом вебе. Методы машинного обучения применяются, чтобы автоматически вывести отображения между схемами. Выведенные отображения хранятся в сети и используются при обработке будущих запросов. Такой подход принят в системе GLUE. Зная две онтологии, GLUE для каждого концепта в одной из них находит самый похожий на него концепт в другой. В системе применяются вероятностные определения нескольких практических мер схожести и несколько стратегий

обучения, в которых используется разнородная информация, содержащаяся как в самих примерах данных, так и в таксономической структуре онтологий. Чтобы повысить точность отображения, GLUE включает общеизвестные знания и ограничения предметной области в процесс отображения схем. Основная идея состоит в том, чтобы предоставить классификаторы концептов. Чтобы вычислить степень схожести между двумя концепами X и Y, данные концепта Y классифицируются с помощью классификатора X, и наоборот. Количество значений, которые удалось успешно классифицировать в каждом направлении, представляет схожесть X и Y.

9.2.3. Отображение на основе общего согласия

При этом подходе узлы, имеющие общий интерес, приходят к согласию об описании общей схемы для разделяемых данных. Общая схема обычно готовится и сопровождается опытными пользователями. В P2P-системе APPA предполагается, что узлы, желающие сотрудничать, например на протяжении некоторого эксперимента, соглашаются на описание общей схемы (Common Schema Description – CSD). При наличии CSD схемы узлов можно задать с помощью представлений. Это похоже на подход ЛКП в системах интеграции данных, только запросы в узлах выражаются в терминах локальных представлений, а не CSD. Еще одно различие между этим подходом и ЛКП заключается в том, что CSD – не глобальная схема, а распространяется на ограниченное множество узлов с общими интересами (рис. 9.9). Таким образом, CSD не вызывает проблем с масштабируемостью. Решая поделиться своими данными, узел должен отобразить свою локальную схему на CSD.

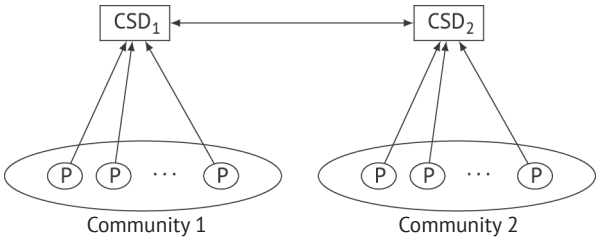


Рис. 9.9 ❖ Отображение схем на основе общего согласия в системе APPA

Пример 9.1. Для CSD с двумя отношениями R_1 и R_2 отображение схемы в узле *p* имеет вид

$p : R(A, B, D) \subseteq csd : R_1(A, B, C), csd : R_2(C, D, E)$.

В этом примере отношение R(A, B, D), которое хочет разделить узел *p*, отображается на отношения $R_1(A, B, C)$, $R_2(C, D, E)$, являющиеся частью CSD. В APPA отображения между CSD и локальными схемами каждого узла хранятся локально в этом узле. Получив запрос *Q* к локальной схеме, узел переформулирует его в запрос к CSD, используя локальные отображения. ♦

9.2.4. Отображение схем методами информационного поиска

В этом подходе отображения схем строятся в момент выполнения запроса с помощью методов информационного поиска (ИП) путем исследования описаний схем, предоставленных пользователями. В системе PeerDB этот подход используется для обработки запросов в неструктурированных P2P-сетях. Описание и атрибуты каждого отношения, разделяемого узлом, хранятся в этом узле. Описания предоставляются пользователями после создания отношений и служат своего рода синонимами имен отношений и атрибутов. Когда пользователь инициирует запрос, формируется и рассылается всем узлам запрос на поиск потенциальных соответствий, а те возвращают соответствующие метаданные. Сопоставляя ключевые слова в метаданных отношений, PeerDB находит отношения, потенциально похожие на отношения в исходном запросе. Найденные отношения предъявляются автору запроса, который решает, продолжить ли выполнение запроса в удаленном узле, владеющем этими отношениями.

В системе Edutella также применяется этот подход для отображения схем в супероднoранговых сетях. Ресурсы в Edutella описываются с помощью RDF-модели метаданных, и описания хранятся в суперузлах. Когда пользователь предъявляет запрос в узле *p*, он посылается суперузлу *p*, который исследует описания хранимых схем и возвращает пользователю адреса релевантных узлов. Если суперузел не находит релевантных узлов, то отправляет запрос другим суперузлам, которые ищут релевантные узлы в своих схемах. Для исследования хранимых схем суперузлы используют язык запросов RDF-QEL, который основан на семантике Datalog и потому совместим со всеми существующими языками запросов и поддерживает функциональность, расширяющую обычные реляционные языки запросов.

9.3. Запросы в P2P-системах

P2P-сети предоставляют базовые методы маршрутизации запросов к релевантным узлам, и этого достаточно для поддержки простых запросов по точному совпадению. Например, как отмечалось выше, технология DHT предлагает механизм для поиска значения по ключу. Но поддержка сложных запросов в P2P-системах, особенно на основе DHT, – трудная проблема, которая недавно привлекла внимание многих исследователей. Основные типы нетривиальных запросов, полезных в P2P-системах, – получение первых k результатов, соединение и запросы по диапазону. В этом разделе мы обсудим методы их обработки.

9.3.1. Получение первых k результатов

Запросы типа «первые k» встречаются во многих областях, например: мониторинг систем и сетей, информационный поиск и мультимедийные базы

данных. В этом случае пользователь запрашивает у системы *k* наиболее релевантных результатов. Степень релевантности (оценка) результата запросу определяется функцией оценивания. Запросы типа «первые k» очень полезны для управления данными в P2P-системах, особенно когда полный набор результатов слишком велик.

Пример 9.2. Рассмотрим P2P-систему для врачей, которые хотят сообща пользоваться некоторой (конфиденциальной) информацией о пациентах для эпидемиологического исследования. Предположим, что все врачи согласовали общее описание отношения `Patient` в реляционном формате. Тогда врача может интересовать получение первых 10 ответов на следующий запрос, ранжированных функцией оценивания по росту и весу:

```
SELECT *
FROM   Patient P
WHERE  P.disease = "diabetes"
AND    P.height < 170
AND    P.weight > 160
ORDER BY scoring-function(height,weight)
STOP AFTER 10
```

Функция оценивания определяет, насколько точно каждый элемент данных соответствует условиям. Например, в приведенном выше запросе функция оценивания могла бы оставить десять человек с наибольшим весом. ♦

Эффективно выполнить запрос типа «первые k» в P2P-системе трудно из-за масштаба сети. В этом разделе мы сначала обсудим самые эффективные методы, предложенные для обработки запросов типа «первые k» в распределенных системах, а затем опишем методы для P2P-систем.

9.3.1.1. Базовые методы

Для обработки запросов типа «первые k» в централизованных и распределенных системах эффективен пороговый алгоритм (Threshold Algorithm – TA). Он применим к запросам, в которых функция оценивания монотонна, т. е. увеличение входного значения никогда не приводит к уменьшению выходного. Многие популярные функции агрегирования, в частности Min, Max и Average, являются монотонными. TA лежит в основе нескольких алгоритмов, которые мы рассмотрим в этом разделе.

Пороговый алгоритм (TA)

В TA предполагается следующая модель, основанная на списках элементов данных, отсортированных по локальным оценкам. Предположим, что имеется *m* списков из *n* элементов каждый таких, что с каждым элементом связана локальная оценка, и эти списки отсортированы по локальным оценкам. Кроме того, у каждого элемента данных имеется общая оценка, вычисляемая по его локальным оценкам во всех списках с помощью заданной функции оценивания. Например, рассмотрим базу данных (т. е. три отсортированных списка) на рис. 9.10. В предположении, что функция оценивания вычисляет сумму локальных оценок одного и того же элемента данных во всех списках, полная оценка элемента d_1 равна 30 + 21 + 14 = 65.

Тогда задача о запросе типа «первые k» заключается в том, чтобы найти k элементов данных с наибольшими оценками. Для этой задачи существует простая и общая модель. Пусть требуется найти первые k кортежей в реляционной таблице относительно некоторой функции оценивания от атрибутов. Для ответа на этот запрос достаточно иметь отсортированный (индексированный) список значений каждого атрибута, участвующего в вычислении функции оценивания, и вернуть k кортежей с наибольшими полными оценками в этих списках. Другой пример – пусть требуется найти первые k документов с наибольшим агрегированным рангом относительно заданного множества ключевых слов. Для ответа на этот запрос нужно для каждого ключевого слова составить ранжированный список документов и вернуть k документов из всех списков с наибольшим агрегированным рангом.

В алгоритме TA применяется два режима доступа к отсортированному списку. Первый – последовательный доступ, при котором каждый элемент рассматривается в порядке появления в списке. Второй – произвольный доступ, когда к каждому элементу можно обратиться напрямую, например с помощью индекса по идентификатору элемента.

Имея m отсортированных списков из n элементов данных каждый, TA (см. алгоритм 9.1) просматривает их параллельно и для каждого элемента получает его локальные оценки во всех списках методом произвольного доступа и вычисляет общую оценку. Кроме того, поддерживается множество Y, содержащее k элементов данных с наибольшими полными оценками на текущий момент. Для остановки TA используется порог, который вычисляется по последним локальным оценкам, встретившимся при последовательном просмотре отсортированных списков. Рассмотрим, например, базу данных на рис. 9.10. В позиции 1 для всех списков (т. е. когда при последовательном доступе мы видели только первые элементы данных) порог равен 30 + 28 + 30 в предположении, что в качестве функции оценивания используется сумма оценок. В позиции 2 порог равен 84. Поскольку элементы списков отсортированы в порядке убывания локальных оценок, порог уменьшается по мере продвижения по спискам. Процесс продолжается, пока не будет найдено k элементов, для которых полные оценки больше порога.

Пример 9.3. Снова рассмотрим базу данных (т. е. три отсортированных списка) на рис. 9.10. Предположим, что запрашиваются первые три элемента (при $k = 3$), а функция оценивания вычисляет сумму локальных оценок элемента во всех списках. Тогда TA сначала рассматривает элементы в позиции 1 каждого списка, т. е. d_1, d_2 и d_3. Он ищет локальные оценки этих элементов в других списках, применяя произвольный доступ, и вычисляет их полные оценки (равные 65, 63 и 70 соответственно). Но ни одна из них не достигает порога в позиции 1 (равного 88). Поэтому в позиции 1 TA не останавливается. В этой позиции мы имеем $Y = \{d_1, d_2, d_3\}$, т. е. k элементов с наибольшими текущими оценками. В позициях 2 и 3 Y будет равно $\{d_3, d_4, d_5\}$ и $\{d_3, d_5, d_8\}$ соответственно. До позиции 6 ни один из элементов Y не имеет полной оценки, большей или равной порогового значения. В позиции 6 пороговое значение равно 63, и это меньше полной оценки каждого из трех элементов в $Y = \{d_3, d_5, d_8\}$. Поэтому TA останавливается. Отметим, что состав Y в позиции 6 точно

такой же, как в позиции 3. Иначе говоря, в позиции 3 Y уже содержит первые k результатов. В этом примере алгоритм TA делает три дополнительных перемещения по каждому списку, которые ничего не привносят в окончательный результат. Это отличительная характеристика алгоритма TA, в котором условие остановки консервативно, из-за чего алгоритм останавливается позже, чем необходимо, – в данном случае он выполняет 9 операций последовательного доступа и 18 = 9 ∗ 2 операций произвольного доступа, которые не дают ничего нового. ♦

Позиция	Список 1		Список 2		Список 3	
	Элемент данных	Локальная оценка s_1	Элемент данных	Локальная оценка s_2	Элемент данных	Локальная оценка s_3
1	d_1	30	d_2	28	d_3	30
2	d_4	28	d_6	27	d_5	29
3	d_9	27	d_7	25	d_8	28
4	d_3	26	d_5	24	d_4	25
5	d_7	25	d_9	23	d_2	24
6	d_8	23	d_1	21	d_6	19
7	d_5	17	d_8	20	d_{13}	15
8	d_6	14	d_3	14	d_1	14
9	d_2	11	d_4	13	d_9	12
10	d_{11}	10	d_{14}	12	d_7	11
...	...	...	...	...	...	...

Рис. 9.10 ❖ Пример базы данных с тремя отсортированными списками

Алгоритмы в духе TA

Для распределенной обработки запросов типа «первые k» предложено несколько алгоритмов в духе TA, т. е. обобщений порогового алгоритма. Мы проиллюстрируем их на примере трехфазного равномерного порогового алгоритма (Three Phase Uniform Threshold – TPUT), который выполняет запросы типа «первые k» за три раунда, предполагая, что каждый список хранится в одном узле (который мы будем называть *владельцем списка*) и что в качестве функции оценивания используется сумма. Алгоритм TPUT, выполняемый инициатором запроса, подробно описан в алгоритме 9.2.

TPUT работает следующим образом:

1) сначала инициатор запроса получает от каждого владельца списка первые k его элементов. Обозначим f функцию оценивания, d полученный элемент данных, а $s_i(d)$ локальную оценку d в списке L_i. Тогда частичная сумма d определяется как $psum(d) = \sum_{m}^{i=1} s_i'(d)$, где $s_i'(d) = s_i(d)$, если d был отправлен координатору владельцем L_i, в противном случае $s_i'(d) = 0$. Инициатор запроса вычисляет частичные суммы для всех полученных элементов и находит элементы с k наибольшими частичными суммами. Частичная сумма k-го элемента данных (называемая *днищем фазы-1*) обозначается λ_1;

Алгоритм 9.1. Пороговый алгоритм (TA)

Вход: $L_1, L_2, \ldots, L_m$: m отсортированных списков по n элементов данных
f : функция оценивания
Выход: Y : список первых k элементов данных
begin
  $j \leftarrow 1$
  threshold $\leftarrow 1$
  min_overall_score $\leftarrow 0$
  while $j \neq n + 1$ *и min_overall_score* < *threshold* **do**
    {последовательный доступ параллельно ко всем m отсортированным спискам}
    for *i от* 1 *до m параллельно* **do**
      {Обработать все элементы в позиции j}
      for *каждого элемента данных d в позиции j списка* L_i **do**
        {прямой доступ к локальным оценкам d в других списках}
        overall_score(d) $\leftarrow f$ (оценки d в каждом L_i)
      end for
    end for
    $Y \leftarrow k$ элементов с текущими наибольшими оценками
    min_overall_score $\leftarrow$ наименьшая полная оценка элементов в Y
    threshold $\leftarrow f$ (локальные оценки элементов в позиции j каждого L_i)
    $j \leftarrow j + 1$
  end while
end

2) инициатор запроса отправляет пороговое значение $\tau = \lambda_1/m$ каждому владельцу списка. В ответ каждый владелец списка отправляет все свои элементы, локальные оценки которых не меньше τ. Интуиция подсказывает, что если на этом этапе о каком-то элементе данных не сообщил ни один узел, то его оценка должна быть меньше λ_1, поэтому он не может входить в состав первых k элементов. Обозначим Y множество элементов данных, полученных от владельцев списков. Инициатор запроса вычисляет новые частичные суммы для элементов, вошедших в Y, и находит элементы с k наибольшими частичными суммами. Частичная сумма k-го элемента данных (называемая *днищем фазы-2*) обозначается λ_2. Пусть верхняя граница оценки элемента d определена как $u(d) = \sum_m^{i=1} u_i(d)$, где $u_i(d) = s_i(d)$, если d был получен, иначе $u_i(d) = \tau$. Для каждого элемента данных $d \in D$, если $u(d)$ меньше λ_2, элемент удаляется из Y. Элементы, оставшиеся в Y, называются кандидатами в первые k, потому что в Y могут присутствовать элементы, полученные не от всех владельцев списков. Для их поиска необходима третья фаза;
3) инициатор запроса посылает множество кандидатов в первые k каждому владельцу списка, который возвращает их оценки. Затем инициатор вычисляет полную оценку, выбирает k элементов данных с наибольшими оценками и возвращает ответ пользователю.

Алгоритм 9.2. Трехфазный равномерный пороговый алгоритм (TPUT)

Вход: $L_1, L_2, \ldots, L_m$: m отсортированных списков по n элементов, принадлежащих разным владельцам
f : функция оценивания
Выход: Y : список первых k элементов данных
begin
  {Фаза 1}
  for *i от* 1 *до m параллельно* **do**
    $Y \leftarrow$ получить первые k элементов данных от владельца списка L_i
  end for
  $Z \leftarrow$ элементы данных с k наибольшими частичными суммами в Y
  $\lambda_1 \leftarrow$ частичная сумма k-го элемента данных в Z
  {Фаза 2}
  for *i от* 1 *до m параллельно* **do**
    отправить λ_1/m владельцу L_i
    $Y \leftarrow$ все элементы данных от владельца L_i, локальные оценки которых не меньше λ_1/m
  end for
  $Z \leftarrow$ элементы данных с k наибольшими частичными суммами в Y
  $\lambda_2 \leftarrow$ частичная сумма k-го элемента данных в Z
  $Y \leftarrow Y -$ {элементы Y, для которых верхняя граница оценки меньше $\lambda 2$}
  {Фаза 3}
  for *i от* 1 *о m параллельно* **do**
    отправить Y владельцу L_i
    $Z \leftarrow$ элементы данных от владельца L_i, принадлежащие одновременно Y и L_i
  end for
  $Y \leftarrow k$ элементов данных с наибольшими полными оценками в Z
end

Пример 9.4. Рассмотрим первые два отсортированных списка (список 1 и список 2) на рис. 9.10. Предположим, что запрашиваются первые два элемента, т. е. $k = 2$, а функцией оценивания является сумма. После фазы 1 порождаются множества $Y = \{d_1, d_2, d_4, d_6\}$ и $Z = \{d_1, d_2\}$. k-й (т. е. второй) элемент – d_2, для которого частичная сумма равна 28. Таким образом, имеем $\lambda_1/2 = 28/2 = 14$. Будем записывать каждый элемент данных d из Y в виде (d, оценка в списке 1, оценка в списке 2). После фазы 2 порождаются множества $Y = \{(d_1, 30, 21), (d_2, 0, 28), (d_3, 26, 14), (d_4, 28, 0), (d_5, 17, 24), (d_6, 14, 27), (d_7, 25, 25), (d_8, 23, 20), (d_9, 27, 23)\}$ и $Z = \{(d_1, 30, 21), (d_7, 25, 25)\}$. Заметим, что вместо d_7 можно было бы взять d_9, поскольку их частичные суммы одинаковы. Таким образом, получаем $\lambda_2/2=50$. Верхние границы оценок элементов Y таковы:

$u(d_1) = 30 + 21 = 51;$
$u(d_2) = 14 + 28 = 42;$
$u(d_3) = 26 + 14 = 40;$
$u(d_4) = 28 + 14 = 42;$
$u(d_5) = 17 + 24 = 41;$
$u(d_6) = 14 + 27 = 41;$

$u(d_7) = 25 + 25 = 50$;
$u(d_8) = 23 + 20 = 43$;
$u(d_9) = 27 + 23 = 50$.

После удаления из Y элементов данных с оценкой верхней границы, меньшей λ_2, имеем $Y = \{d_1, d_7, d_9\}$. В этом случае третья фаза необязательна, потому что у всех элементов данных есть локальные оценки. Поэтому окончательный результат равен $Y = \{d_1, d_7\}$, или $Y = \{d_1, d_9\}$. ♦

Если количество списков m велико, то время ответа алгоритма TPUT намного меньше, чем у базового алгоритма TA.

Алгоритм лучшей позиции (BPA)

Есть немало примеров баз данных, для которых алгоритм TA продолжает просматривать списки, хотя все первые k результатов уже получены (как в примере 9.3) и можно было бы остановить работу гораздо раньше. Исходя из этого наблюдения, были предложены алгоритмы лучшей позиции (best position algorithm – BPA), которые выполняют запросы типа «первые k» гораздо эффективнее, чем TA. Основная идея BPA заключается в том, чтобы рассматривать в механизме остановки специальные позиции, которые называются *лучшими*. Интуитивно лучшей является позиция с максимальным индексом такая, что любая позиция до нее также уже встречалась. Условие остановки основано на полной оценке, вычисленной с помощью лучших позиций во всех списках.

Базовая версия BPA (см. алгоритм 9.3) похожа на TA, но имеет следующие отличия: запоминаются все позиции, которые встречались при последовательном или произвольном доступе, вычисляются лучшие позиции и изменено условие остановки. Для каждого списка L_i обозначим P_i множество позиций, которые встречались при последовательном или произвольном доступе к L_i. Обозначим bp_i (лучшую позицию в L_i) позицию с наибольшим индексом в P_i такую, что любая позиция в L_i между 1 и bp_i также принадлежит P_i. Иными словами, bp_i лучшая, потому что мы уверены, что все позиции в L_i между 1 и bp_i встречались при последовательном или произвольном доступе. Обозначим $s_i(bp_i)$ локальную оценку элемента данных в позиции bp_i списка L_i. Тогда порог в алгоритме BPA равен $f(s_1(bp_1), s_2(bp_2), \ldots, s_m(bp_m))$ для некоторой функции f.

Пример 9.5. Для иллюстрации базового алгоритма BPA снова рассмотрим три отсортированных списка на рис. 9.10 и запрос Q из примера 9.3.

1. В позиции 1 BPA видит элементы данных d_1, d_2 и d_3. Для каждого из них он производит прямой доступ и получает его локальные оценки и позиции во всех списках. Следовательно, на этом шаге в списке L_1 были просмотрены позиции 1, 4 и 9, в которых находятся соответственно элементы d_1, d_3 и d_2. Таким образом, $P_1 = \{1, 4, 9\}$, и лучшая позиция в L_1 – это $bp_1 = 1$ (поскольку следующая позиция 4, а позиции 2 и 3 еще не встречались). Для L_2 и L_3 имеем $P_2 = \{1, 6, 8\}$ и $P_3 = \{1, 5, 8\}$, так что $bp_2 = 1$ и $bp_3 = 1$. Стало быть, полная оценка по лучшим позициям равна $\lambda = f(s_1(1), s_2(1), s_3(1)) = 30 + 28 + 30 = 88$. В позиции 1 множество трех элемен-

тов данных с наивысшими оценками $Y = \{d_1, d_2, d_3\}$, и поскольку полная оценка этих элементов меньше λ, BPA еще не может остановиться.

2. В позиции 2 BPA видит элементы d_4, d_5 и d_6. Таким образом, $P_1 = \{1, 2, 4, 7, 8, 9\}$, $P_2 = \{1, 2, 4, 6, 8, 9\}$ и $P_3 = \{1, 2, 4, 5, 6, 8\}$. Следовательно, имеем $bp_1 = 2$, $bp_2 = 2$ и $bp_3 = 2$, так что $\lambda = f(s_1(2), s_2(2), s_3(2)) = 28 + 27 + 29 = 84$. Полная оценка элементов данных, входящих во множество $Y = \{d_3, d_4, d_5\}$, меньше 84, поэтому BPA не останавливается.
3. В позиции 3 BPA видит элементы d_7, d_8 и d_9. Следовательно, $P_1 = P_2 = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ и $P_3 = \{1, 2, 3, 4, 5, 6, 7, 8, 10\}$. Поэтому имеем $bp_1 = 9$, $bp_2 = 9$ и $bp_3 = 8$. Полная оценка по лучшим позициям равна $\lambda = f(s_1(9), s_2(9), s_3(8)) = 11+13+14 = 38$. В этой позиции $Y = \{d_3, d_5, d_8\}$. Поскольку оценки всех элементов данных, принадлежащих Y, больше λ, BPA останавливается – именно в той позиции, в которой BPA впервые получил все первые k результатов.

Напомним, что при работе с этой базой данных TA останавливается в позиции 6. ♦

Алгоритм 9.3. Алгоритм лучшей позиции (BPA)

Вход: $L_1, L_2, \ldots, L_m$: m отсортированных списков по n элементов
f : функция оценивания
Выход: Y : список первых k элементов данных
begin
    $j \leftarrow 1$
    threshold $\leftarrow 1$
    min_overall_score $\leftarrow 0$
    for *i от* 1 *до m параллельно* **do**
        $P_i \leftarrow \emptyset$
    end for
    while $j \neq n + 1$ *и min_overall_score* < *threshold* **do**
        {последовательный доступ параллельно ко всем m отсортированным спискам}
        for *i от* 1 *до m параллельно* **do**
            {Обработать все элементы в позиции j}
            for *каждого элемента данных d в позиции j списка* L_i **do**
                {прямой доступ к локальным оценкам d в других списках}
                overall_score(d) $\leftarrow f$ (оценки d в каждом L_i)
            end for
            $P_i \leftarrow P_i \cup$ {позиции, встречавшиеся при последовательном или произвольном доступе}
            $bp_i \leftarrow$ лучшая позиция в L_i
        end for
        $Y \leftarrow k$ элементов с текущими наибольшими оценками
        min_overall_score $\leftarrow$ наименьшая общая оценка элементов в Y
        threshold $\leftarrow f$ (локальные оценки элементов в позициях bp_i каждого L_i)
        $j \leftarrow j + 1$
    end while
end

Показано, что для любого множества отсортированных списков BPA останавливается не позже, чем TA, а стоимость его выполнения никогда не бывает выше, чем для TA. Также показано, что стоимость выполнения BPA может быть в m – 1 раз (где m – количество отсортированных списков) меньше, чем для TA. Хотя алгоритм BPA весьма эффективен, он все равно делает лишнюю работу. В частности, BPA (как и TA) может несколько раз обращаться к некоторым элементам данных при последовательном просмотре разных списков. Например, элемент, который встречался в некоторой позиции во время последовательного просмотра одного списка и к которому, следовательно, производился произвольный доступ в других списках, может снова встретиться во время последовательного просмотра других списков. Улучшенный алгоритм BPA2 избегает таких ситуаций и, значит, работает гораздо эффективнее BPA. Он не передает ранее встречавшиеся позиции от владельцев списков инициатору запроса. Поэтому инициатор может не хранить встречавшиеся позиции и их локальные оценки. Кроме того, каждая позиция в списке обрабатывается не более одного раза. Количество обращений к спискам в BPA2 может быть в m – 1 раз меньше, чем в BPA.

9.3.1.2. Запросы типа «первые k» в неструктурированных системах

Один из возможных подходов к обработке запросов типа «первые k» в неструктурированных системах – разослать запрос всем узлам, собрать все ответы, ранжировать их с помощью функции оценивания и вернуть пользователю k ответов с наивысшими оценками. Однако этот подход неэффективен с точки зрения времени ответа и затрат на передачу данных.

Первое эффективное решение было предложено в неструктурированной P2P-системе PlanetP. В ней адресуемая по содержимому служба публикации-подписки реплицирует данные между P2P-сообществами численностью до десяти тысяч узлов. Алгоритм обработки запроса типа «первые k» работает следующим образом. Инициатор запроса Q ранжирует узлы по релевантности ответа запросу, обращается к ним поочередно в порядке убывания ранга и просит вернуть множество элементов с наибольшей оценкой, а также их оценки. Для вычисления релевантности узлов используется глобальный полностью реплицированный индекс, содержащий соответствия между термами и узлами. Этот алгоритм очень хорошо работает в системах умеренного масштаба. Но в больших P2P-системах поддержание реплицированного индекса в актуальном состоянии вступает в противоречие с масштабируемостью.

Мы опишем еще одно решение, разработанное в контексте APPA – независимой от сети P2P-системы управления данными. Была предложена полностью распределенная инфраструктура для выполнения запросов типа «первые k», принимающая во внимание возможность изменения состава узлов во время выполнения запроса, а также ситуации, когда некоторые узлы покидают систему, не завершив обработку запроса. Если имеется запрос Q типа «первые k» с заданным временем жизни TTL, то базовый алгоритм, который называется Fully Decentralized Top-k (FD), работает следующим образом (см. алгоритм 9.4):

Алгоритм 9.4. Fully Decentralized Top-k (FD)

```
Вход: Q: запрос типа первые k
f : функция оценивания
TTL: время жизни
w: время ожидания
Выход: Y : список первых k элементов данных
begin
    На стороне инициатора запроса
    begin
        отправить Q соседям
        Final_score_list ← объединить локальные списки оценок, полученные
        от соседей
        for каждого узла p в Final_score_list do
            Y ← получить первые k элементов данных от p
        end for
    end
    for каждого узла, получившего запрос Q от узла p do
        TTL ← TTL − 1
        if TTL > 0 then
            послать Q соседям
        end if
        Local_score_list ← извлечь первые k локальных оценок
        ожидать в течение времени w
        Local_score_list ← Local_score_list ∪ первые k полученных оценок
        отправить Local_score_list узлу p
    end for
end
```

1) **отправка запроса.** Инициатор запроса рассылает Q доступным узлам, количество переходов до которых меньше TTL;
2) **локальное выполнение запроса и ожидание.** Каждый узел p, получивший запрос Q, выполняет его локально: находит локальные элементы данных, которые удовлетворяют предикату запроса, применяет к ним функцию оценивания, выбирает первые k элементов и локально сохраняет их вместе с оценками. Затем p ожидает получения результатов от соседа. Но поскольку некоторые соседи могли покинуть P2P-системы, не прислав p список оценок, время ожидания имеет предел, который вычисляется для каждого узла в зависимости от полученного TTL, параметров сети и локальных параметров обработки в данном узле;
3) **объединение и возврат.** На этом этапе наивысшие оценки следующим образом передаются инициатору запроса с помощью основанного на дереве алгоритма. По истечении времени ожидания p объединяет свои k локальных наивысших оценок с полученными от соседей и посылает результат своему родителю (узлу, от которого получил запрос Q) в виде списка оценок. Для минимизации сетевого трафика алгоритм FD не передает наверх сами элементы (которые могут занимать много места), а только оценки и адреса. Список оценок – это просто список

k пар (*a*, *s*), где *a* – адрес узла, владеющего элементом данных, а *s* – оценка этого элемента;

4) **получение данных.** Получив списки оценок от своих соседей, инициатор запроса формирует окончательный список оценок, объединяя свои *k* лучших локальных оценок со списками, полученными от соседей. Затем он запрашивает первые *k* элементов данных непосредственно у их владельцев.

Алгоритм полностью распределенный и не зависит от существования каких-то специальных узлов, поэтому может примириться с изменением множества узлов во время выполнения. В частности, решены следующие проблемы: узел становится недоступным на этапе объединения и возврата; узел, владеющий элементами данных с лучшими оценками, становится недоступным на этапе получения данных; получение списков оценок от узла после истечения времени ожидания. Исследования производительности FD показывают, что он дает значительный выигрыш в терминах стоимости коммуникации и времени ответа.

9.3.1.3. Запросы типа «первые k» в DHT-системах

Как мы уже говорили, основное назначение DHT – отобразить множество ключей на узлы P2P-системы и эффективно находить узел, отвечающий за данный ключ. Поэтому они предлагают эффективную и масштабируемую поддержку запросов с точным совпадением. Однако поддержать запрос типа «первые k» в системе на основе DHT нелегко. Простое решение – выбрать все кортежи из всех отношений, участвующих в запросе, вычислить оценку каждого кортежа и вернуть *k* кортежей с наибольшими оценками. Однако такое решение не масштабируется на случай большого числа хранимых кортежей. Еще одно решение – хранить все кортежи одного отношения под одним ключом (например, им может быть имя отношения), так чтобы все кортежи хранились в одном узле. Тогда обработку запроса типа «первые k» можно произвести в центральном узле, пользуясь хорошо известными централизованными алгоритмами. Но при этом такой узел становится узким местом и точкой общего отказа.

Решение, предложенное в проекте APPA, основано на алгоритме TA (см. раздел 9.3.1.1) и механизме хранения разделяемых данных в DHT-таблице полностью распределенным способом. В APPA узлы могут хранить свои кортежи в DHT одним из двух взаимно дополняющих методов: хранение кортежей и хранение значений атрибутов. В первом случае каждый кортеж хранится в DHT под ключом, совпадающим с его идентификатором (например, первичным ключом). Это позволяет искать кортеж по идентификатору, как в первичном индексе. Во втором случае в DHT по отдельности хранятся атрибуты, которые могут встречаться в предикате сравнения на равенство или в функции оценивания. Таким образом, кортежи можно искать по значениям атрибутов, как при использовании вторичного индекса. У хранения значений атрибутов есть два важных свойства: (1) после получения значения атрибута из DHT узлы могут легко извлечь соответствующий кортеж; (2) относительно «близкие» значения атрибутов хранятся в одном узле. Чтобы

обеспечить выполнение первого свойства, вместе со значением атрибута хранится ключ, с помощью которого можно получить весь кортеж. Второе свойство гарантируется благодаря понятию доменного секционирования следующим образом. Рассмотрим атрибут a, и пусть D_a – область его значений. Предположим, что на D_a определен полный порядок $\prec$ (например, D_a – множество чисел). D_a разбивается на n непустых поддоменов $d_1, d_2, \ldots, d_n$, так что их объединение равно D_a, никакие два поддомена не пересекаются и для каждого $v_1 \in d_i$ и $v_2 \in d_j$, если $i < j$, то $v_1 \prec v_2$. Функция хеширования применяется к поддомену значений атрибута. Таким образом, если значения атрибута попадают в один поддомен, то ключ хранения будет одинаковым и оба значения хранятся в одном узле. Чтобы избежать асимметрии при хранении атрибутов (т. е. асимметричного распределения их значений между поддоменами), доменное секционирование производится таким образом, чтобы значения атрибутов были равномерно распределены между поддоменами. Для этого используется основанная на гистограммах информация, описывающая распределение значений.

При такой модели хранения алгоритм обработки запроса типа «первые k», называемый DHTop (см. алгоритм 9.5), работает следующим образом. Пусть Q – запрос типа «первые k», f – его функция оценивания, а p_0 – узел, инициировавший Q. Для простоты предположим, что функция f монотонна. Назовем атрибутами оценивания набор атрибутов, передаваемых функции оценивания в качестве аргументов. DHTop начинает работу в узле p_0 и состоит из двух этапов: сначала готовятся упорядоченные списки поддоменов-кандидатов, а затем из них извлекаются значения атрибутов и соответствующие кортежи, пока не будут получены первые k кортежей. Ниже описаны детали алгоритма.

1. Для каждого атрибута оценивания $\mathtt{A}_i$ узел p_0 готовит список поддоменов и сортирует их в порядке убывания их положительного влияния на функцию оценивания. Из каждого списка p_0 удаляет поддомены, ни один член которых не удовлетворяет условиям Q. Например, если согласно условию атрибут оценивания должен быть равен некоторой константе (например, $\mathtt{A}_i = 10$), то p_0 удаляет из списка все поддомены, кроме того, которому принадлежит эта константа. Обозначим $L_{\mathtt{A}i}$ список, подготовленный на этом этапе для атрибута оценивания $\mathtt{A}_i$.
2. Для всех атрибутов оценивания $\mathtt{A}_i$ узел p_0 параллельно выполняет следующие действия. Он посылает Q и $\mathtt{A}_i$ узлу, скажем p, который отвечает за хранение всех значений из первого поддомена $L_{\mathtt{A}i}$, и просит его вернуть значения $\mathtt{A}_i$ в p. Значения возвращаются узлу p_0 в порядке их положительного влияния на функцию оценивания. Получив значение атрибута, p_0 извлекает соответствующий кортеж, вычисляет его оценку и сохраняет, если оценка входит во множество первых k вычисленных на данный момент наивысших оценок. Этот процесс продолжается, пока не будет получено k кортежей с оценками больше порога, вычисленного на основе полученных к текущему моменту значений атрибута. Если значений атрибута, которые p вернул p_0, недостаточно для определения первых k кортежей, то p_0 посылает Q и $\mathtt{A}_i$ узлу, отвечающему за второй поддомен $L_{\mathtt{A}i}$, и т. д., пока не будут найдены первые k кортежей.

Алгоритм 9.5. Первые k в DHT (DHTop)

```
Вход: Q: запрос типа «первые k»;
f : функция оценивания;
A: набор m атрибутов, используемых в f
Выход: Y : список первых k кортежей
begin
    {Этап 1: подготовить списки поддоменов атрибутов}
    for каждого атрибута оценивания A_i в A do
        L_Ai ← все поддомены A_i
        L_Ai ← L_Ai – поддомены, не удовлетворяющие условию Q
        Отсортировать L_Ai в порядке возрастания поддоменов
    end for
    {Этап 2: извлекать значения атрибутов и соответствующие кортежи,
      пока не будут получены первые k кортежей}
    Done ← false
    for каждого атрибута оценивания A_i в A параллельно do
        i ← 1
        while (i < число поддоменов A) и не Done do
            отправить Q узлу p, в котором хранятся значения атрибутов
              поддомена i из L_Ai
            Z ← значения A_i (в порядке убывания) из узла p, удовлетворяющие
              условию Q вместе с ключами хранения данных
            for каждого полученного значения v do
                получить кортеж v
                Y ← k кортежей с наивысшими текущими оценками
                threshold ← f (v_1, v_2, ..., v_m), где v_i – последнее значение,
                  полученное для атрибута A_i в A
                min_overall_score ← наименьшая полная оценка кортежей в Y
                if min_overall_score ⩽ threshold then
                    Done ← true
                end if
                i ← i + 1
            end for
        end while
    end for
end
```

Пусть A_1, A_2, ..., A_m – атрибуты оценивания, а v_1, v_2, ..., v_m – их последние полученные значения. По определению, порог равен $\tau = f(v_1, v_2, ..., v_m)$. Основное свойство алгоритма DHTop заключается в том, что после получения каждого нового значения атрибута порог уменьшается. Поэтому после получения некоторого количества атрибутов и их кортежей количество кортежей с оценкой больше порога превысит k, и алгоритм остановится. Математически доказано, что DHTop правильно работает для монотонных функций оценивания, а также для большой группы немонотонных функций.

9.3.1.4. Запросы типа «первые k» в супероднoранговых системах

Типичный алгоритм обработки запросов типа «первые k» в супероднoранговых системах – тот, что используется в системе Edutella. В этой системе имеется относительно немного суперузлов, и предполагается, что все они высокодоступны и располагают большой вычислительной мощностью. Суперузлы отвечают за обработку запросов типа «первые k», а остальные узлы только выполняют запросы локально и оценивают свои собственные ресурсы. Алгоритм очень прост и работает следующим образом. Инициатор посылает запрос *Q* своему суперузлу, а тот рассылает его другим суперузлам. Суперузлы направляют *Q* релевантным подведомственным узлам. Каждый узел, владеющий какими-то данными, релевантными *Q*, оценивает их и отправляет элемент данных с максимальной оценкой своему суперузлу. Каждый суперузел выбирает элемент с максимальной оценкой из всех полученных им. Чтобы определить второй лучший элемент, он запрашивает у узла, приславшего первый, и только у него элемент со следующей по порядку оценкой. Суперузел сравнивает вновь полученный элемент со вторым из полученных ранее и выбирает из них наилучший. Затем он обращается к узлу, приславшему второй лучший элемент, и т. д., пока не будут получены все *k* лучших элементов. Наконец, суперузлы отправляют свои лучшие элементы суперузлу узла-инициатора, который выбирает из них первые *k* и возвращает инициатору запроса. Этот алгоритм минимизирует обмен данными между узлами и суперузлами, поскольку, получив элементы данных с максимальной оценкой от подведомственных узлов, каждый суперузел запрашивает следующий элемент только у одного узла.

9.3.2. Запросы с соединением

Самые эффективные алгоритмы соединения в распределенных и параллельных базах данных основаны на хешировании. Поэтому тот факт, что DHT опирается на хеширование для хранения и поиска данных, можно естественным образом использовать для эффективной поддержки запросов с соединением. Базовое решение было предложено в P2P-системе PIER, которая поддерживает сложные запросы поверх DHT. Это вариация на тему параллельного алгоритма хеш-соединения PHJ (см. раздел 8.4.1), который мы будем называть PIERjoin. Как и PHJ, алгоритм PIERjoin предполагает, что у соединяемых отношений и у результирующего отношения есть дом (который в PIER называется *пространством имен*) – это узлы, в которых хранятся горизонтальные фрагменты отношения. Затем он пользуется методом put, чтобы распределить кортежи по множеству узлов, так чтобы кортежи с одинаковым значением атрибута соединения оказались в одном узле. Для локального выполнения соединения в PIER реализована версия симметричного алгоритма хеш-соединения (см. раздел 8.4.1.2), которая эффективно поддерживает конвейерный параллелизм. При симметричном хеш-соединении двух отношений каждый узел, получивший подлежащие соединению кортежи, хранит

две хеш-таблицы, по одной на каждое отношение. Получив новый кортеж из того или другого отношения, узел добавляет его в соответствующую хеш-таблицу и апробирует относительно другой хеш-таблицы на основе полученных к этому моменту кортежей. PIER опирается на DHT также для того, чтобы справиться с динамическим поведением узлов (присоединением к сети или выходом из нее во время выполнения запроса), поэтому гарантий полноты результатов этот алгоритм не дает.

Запрос Q с бинарным соединением (и, возможно, предикатами выборки) PIERjoin обрабатывает в три этапа (см. алгоритм 9.6): многоадресная рассылка, хеширование и апробирование-соединение.

1. **Этап многоадресной рассылки.** Инициатор запроса рассылает Q всем узлам, в которых хранятся кортежи соединяемых отношений R и S, т. е. их домам.
2. **Этап хеширования.** Каждый узел, получивший запрос Q, просматривает свое локальное отношение в поисках кортежей, удовлетворяющих предикату выборки (если таковой задан). Затем он отправляет выбран-

Алгоритм 9.6. PIERjoin

Вход: Q: запрос с соединением отношений R и S по атрибуту A;
h: хеш-функция;
H_R, H_S: дома R и S
Выход: T: результат соединения;
H_T: дом T
begin
  {этап многоадресной рассылки}
  Узел, инициировавший запрос, отправляет Q всем узлам из H_R и H_S
  {Этап хеширования}
  for *каждого узла p в* H_R*, получившего Q, параллельно* **do**
    for *каждого кортежа* r *в* R_p*, который удовлетворяет предикату выборки,* **do**
      разместить r с использованием $h(H_T, A)$
    end for
  end for
  for *каждого узла p в* H_S*, получившего Q, параллельно* **do**
    for *каждого кортежа* s *в* S_p*, который удовлетворяет предикату выборки,* **do**
      разместить s с использованием $h(H_T, A)$
    end for
  end for
  {Этап апробирования-соединения}
  for *каждого узла p в* H_T *параллельно* **do**
    if *поступил новый кортеж i* **then**
      if *i – это кортеж* r **then**
        апробировать кортежи s в S_p с применением $h(A)$
      else
        апробировать кортежи r в R_p с применением $h(A)$
      end if
      $T_p \leftarrow r \bowtie s$
    end if
  end for
end

ные кортежи в дом результирующего отношения, пользуясь операцией `put`. Ключ DHT, который передается `put`, вычисляется на основе дома результирующего отношения и атрибута соединения.

3. **Этап апробирования-соединения.** Каждый узел в доме результирующего отношения, получив новый кортеж, вставляет его в соответствующую хеш-таблицу, апробирует противоположную хеш-таблицу, чтобы найти кортежи, соответствующие предикату соединения (и предикату выборки, если он задан), и строит результирующие соединенные кортежи. Напомним, что «дом» горизонтального секционированного отношения был определен в главе 4 как множество узлов, в которых хранятся секции. В данном случае секционирование производится хешированием по атрибуту соединения. Домом результата также является секционированное отношение (построенное с помощью операций `put`), поэтому оно размещено в нескольких узлах.

Этот базовый алгоритм можно улучшить в разных направлениях. Например, если одно отношение уже хешировано по атрибутам соединения, то его дом можно использовать как дом результата, применив вариант параллельного алгоритма ассоциативного соединения (PAJ) (см. раздел 8.4.1), в котором только одно отношение нужно хешировать и рассылать через DHT.

9.3.3. Запросы по диапазону

Напомним, что в запросах по диапазону фраза **`WHERE`** имеет вид «атрибут `A` принадлежит диапазону [*a*, *b*]», где *a* и *b* – числа. Структурированные P2P-системы, в частности основанные на DHT, очень эффективно поддерживают запросы на точное совпадение (вида «`A` = *a*»), но испытывают трудности с запросами по диапазону. Основная причина в том, что хеширование уничтожает упорядочение данных, без которого сложно находить диапазоны.

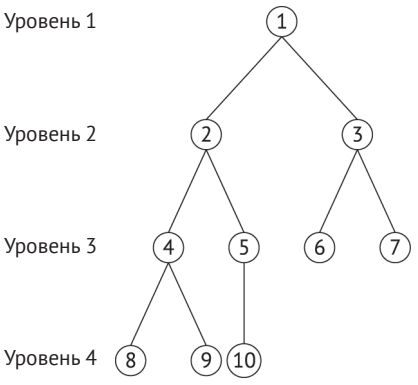
Существует два подхода к поддержке запросов по диапазону в структурированных P2P-системах: дополнить DHT свойствами близости или сохранения порядка либо поддержать упорядочение ключей в структурах на основе деревьев. Первый подход успешно применен в нескольких системах. Локально-чувствительное хеширование – этот обобщение DHT, при котором похожие диапазоны с высокой вероятностью хешируются в один узел DHT. Однако этот метод дает только приближенные ответы и может создавать несбалансированную нагрузку в больших сетях.

Префиксное хеш-дерево (Prefix Hash Tree – PHT) – древовидная распределенная структура данных, которая поддерживает запросы по диапазону к DHT, просто используя операцию поиска в DHT. В роли индексируемых данных выступают строки длины *D*. У каждого узла имеется 0 или 2 дочерних узла, а ключ *k* хранится в листовом узле, метка которого является префиксом *k*. Кроме того, листовые узлы связаны со своими соседями. Операция поиска по ключу *k* в алгоритме PHT должна вернуть уникальный листовый узел *leaf*(*k*), метка которого является префиксом *k*. У ключа *k* длины *D* имеется *D* + 1 префиксов. Найти *leaf*(*k*) можно путем линейного просмотра этих потенциальных *D* + 1 узлов. Но поскольку PHT – двоичное дерево, линейный

просмотр можно заменить двоичным поиском по длине префикса. Это сокращает количество обращений к DHT с $D + 1$ до $\log D$. Для заданных ключей a и b таких, что $a \leqslant b$, поддерживается два алгоритма обработки запросов по диапазону, основанных на поиске в PHT. Первый – последовательный: ищется *leaf*(a), а затем последовательно просматривается связный список листовых узлов, до тех пор пока не будет найден *leaf*(b). Второй – параллельный: сначала ищется узел, соответствующий наименьшему диапазону префиксов, полностью покрывающему диапазон [a, b]. Чтобы найти этот узел, применяется простой поиск в DHT, и запрос рекурсивно переадресуется тем дочерним узлам, которые перекрываются с [a, b].

Как и любая схема хеширования, первый подход может приводить к асимметрии данных, из-за которой в узлах оказываются несбалансированные диапазоны, что негативно сказывается на балансировке нагрузки. Для решения этой проблемы во втором подходе используются основанные на деревьях структуры для поддержания сбалансированных диапазонов ключей. Первой попыткой построить P2P-сеть, основанную на структуре сбалансированного дерева, стала система BATON (BAlanced Tree Overlay Network). Ниже мы подробнее опишем BATON и то, как в ней устроена поддержка запросов по диапазону.

В BATON узлы организованы в виде сбалансированного двоичного дерева (каждый узел дерева обслуживается узлом). Позиция узла в BATON определяется кортежем (уровень, номер), причем уровни нумеруются с 0 (корень), а номера присваиваются в порядке симметричного (in-order) обхода дерева, начиная с корня, которому присвоен номер 1. В каждом узле дерева хранятся ссылки на родителя, дочерние узлы, смежные узлы и избранные узлы на том же уровне. Последние хранятся в двух таблицах: *левой* и *правой таблицах маршрутизации*. Для узла с номером i в этих таблицах находятся ссылки на узлы того же уровня с номерами, меньшими (левая таблица) или большими (правая таблица) i на степень 2; j-й элемент левой (правой) таблицы маршрутизации в узле i содержит ссылку на узел с номером $i - 2^{j-1}$ (соответственно $i + 2^{j-1}$) на том же уровне дерева. На рис. 9.11 показана таблица маршрутизации в узле 6.



Узел 6: уровень 2, номер=3
parent=3, leftchild=null, rightchild=null
leftadjacent=1, rightadjacent=3

Левая таблица маршрутизации

	Узел	Левый дочерний узел	Правый дочерний узел	Нижняя граница	Верхняя граница
0	5	10	null	LB5	UB5
1	4	8	9	LB4	UB4

Правая таблица маршрутизации

	Узел	Левый дочерний узел	Правый дочерний узел	Нижняя граница	Верхняя граница
0	7	null	null	LB7	UB7

Рис. 9.11 ❖ Система BATON: древовидный индекс и таблицы маршрутизации в узле 6

В BATON каждому листовому и внутреннему узлу назначается диапазон значений. Этот диапазон ассоциирован с каждой ссылкой в таблице маршрутизации, и когда диапазон изменяется, модифицируется содержимое элемента таблицы. Диапазон значений, управляемый узлом, обязан располагаться справа от диапазона, управляемого левым поддеревом узла, и слева от диапазона, управляемого его правым поддеревом (см. рис. 9.12). Таким образом, BATON строит эффективную распределенную индексную структуру. Присоединение и выход узлов обрабатываются так, что дерево остается сбалансированным, – запрос поднимается вверх по дереву, когда узел присоединяется к сети, и вниз, когда узел покидает сеть; таким образом, для дерева, содержащего n узлов, требуется не более чем $O(\log n)$ шагов.

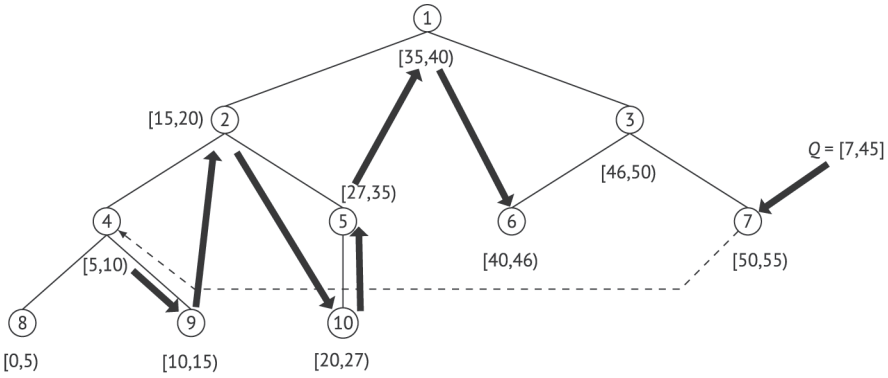


Рис. 9.12 ❖ Обработка запроса по диапазону в BATON

Запрос по диапазону обрабатывается следующим образом (алгоритм 9.7). Для запроса по диапазону $[a, b]$, инициированного узлом i, система ищет узел, в который попадает нижняя граница диапазона. Этот узел локально находит кортежи, принадлежащие указанному диапазону, и переадресует запрос своему правому смежному узлу. Вообще, каждый узел, получивший запрос, проверяет свои локальные кортежи и обращается к своему правому смежному узлу, пока не встретится узел, содержащий верхнюю границу диапазона. Частичные ответы, полученные до момента нахождения точки пересечения, отправляются инициатору запроса. Для нахождения первого пересечения требуется $O(\log n)$ шагов алгоритма поиска точного совпадения. Поэтому для получения ответа на запрос по диапазону необходимо $O(\log n + X)$ шагов, где X – количество узлов, покрывающих указанный диапазон.

Пример 9.6. Рассмотрим запрос Q по диапазону [7, 45], инициированный в узле 7 на рис. 9.12. Сначала BATON выполняет запрос на точное совпадение в поисках узла, содержащего нижнюю границу диапазона (штриховая линия на рисунке). Нижняя граница лежит в диапазоне, хранящемся в узле 4, этот узел локально проверяет кортежи на принадлежность к диапазону и переадресует запрос своему правому смежному узлу (узел 9). Узел 9 проверяет свои локальные кортежи на принадлежность к диапазону и переадресует

запрос узлу 2. Узлы 10, 5, 1 и 6 получают запрос, проверяют локальные кортежи и обращаются к своим правым смежным узлам, пока не встретится узел, содержащий правую границу диапазона. ♦

Алгоритм 9.7. BatonRange

```
Вход: Q: запрос по диапазону [a, b]
Выход: T: результирующее отношение
begin
    {Поиск узла, в котором хранится нижняя граница диапазона}
    На стороне узла инициатора
    begin
        найти узел p, где хранится a
        отправить Q узлу p
    end
    for каждого узла p, получившего Q do
        T_p ← Range(p) ∩ [a, b]
        отправить T_p инициатору запроса
        if Range(RightAdjacent (p)) ∩ [a, b] ≠ ∅ then
            положить p равным правому смежному с p узлу
            отправить Q узлу p
        end if
    end for
end
```

9.4. Согласованность реплик

Чтобы повысить доступность данных и производительность доступа, данные в P2P-системах реплицируются. Однако разные P2P-системы обеспечивают сильно различающиеся уровни согласованности реплик. В ранних простых P2P-системах, в частности Gnutella и Kazaa, данные были только статическими (например, музыкальные файлы), а репликация была «пассивной», т. е. возникала естественным образом по мере того, как узлы запрашивали файлы друг у друга и копировали их к себе (по существу, это кеширование данных). В более сложных P2P-системах, где возможно обновление реплик, возникает необходимость в полноценном механизме управления репликами. К сожалению, большая часть работ по согласованности реплик относится только к DHT. Можно выделить три подхода: базовая поддержка в DHT, актуальность данных в DHT и урегулирование реплик. В этом разделе мы познакомимся с основными методами, используемыми в этих подходах.

9.4.1. Базовая поддержка в DHT

Для улучшения доступности данных большинство DHT полагаются на репликацию – хранят пары (*ключ*, *данные*) в нескольких узлах, например используя несколько хеш-функций. Если один узел недоступен, то его данные

все-таки можно получить от других узлов, где хранятся реплики. Некоторые DHT предоставляют базовую поддержку, позволяющую приложению позаботиться о согласованности реплик. В этом разделе мы опишем методы, применяемые в двух популярных DHT: CAN и Tapestry.

В CAN имеется два подхода к поддержке репликации. Первый – использовать m хеш-функций для отображения одного ключа на m точек в координатном пространстве, т. е. реплицировать одну пару (*ключ*, *данные*) на m разных узлов сети. Второй представляет собой оптимизацию базового устройства CAN – узел заранее раздает популярные ключи своим соседям, обнаружив, что перегружен запросами на эти ключи. При таком подходе с реплицированными ключами необходимо ассоциировать поле TTL, чтобы автоматически отменить эффект репликации по завершении периода повышенной нагрузки. Кроме того, предполагается, что данные неизменяемые (допускают только чтение).

Tapestry – расширяемая P2P-система, предоставляющая децентрализованные службы поиска объектов и маршрутизации поверх структурированной наложенной сети. Она маршрутизирует сообщения до логических оконечных точек (т. е. оконечных точек, идентификаторы которых не ассоциированы ни с каким физическим местоположением), например узлов или реплик объектов. Это открывает возможность доставки сообщений до мобильных или реплицированных оконечных точек в условиях нестабильности базовой инфраструктуры. Кроме того, Tapestry учитывает задержку с целью определить окрестность узла. Механизмы поиска объектов и маршрутизации в Tapestry работают следующим образом. Пусть o – объект с идентификатором $id(o)$; вставка o в P2P-сеть затрагивает два узла: серверный узел (обозначаемый n_s), владеющий o, и корневой узел (обозначаемый n_r), в котором хранится отображение вида $(id(o), n_s)$, показывающее, что объект с идентификатором $id(o)$ находится в узле n_s. Корневой узел динамически определяется глобально согласованным детерминированным алгоритмом. На рис. 9.13a показано, что при вставке o в узел n_s тот публикует $id(o)$ на своем корневом узле, направляя из n_s в n_r сообщение, содержащее отображение $(id(o), n_s)$. Это отображение сохраняется во всех узлах вдоль пути следования сообщения. В процессе обработки запроса поиска, например «$id(o)$?» на рис. 9.13a, сообщение с запросом $id(o)$ первоначально направляется в n_r, но может быть остановлено раньше, если будет найден узел, содержащий отображение $(id(o), n_s)$. Для маршрутизации сообщения до корня $id(o)$ каждый узел переправляет его своему соседу с логическим идентификатором, больше всего похожим на $id(o)$.

Tapestry предлагает цельную инфраструктуру, необходимую для полезного использования реплик, она показана на рис. 9.13b. Каждая вершина графа представляет узел P2P-сети и содержит логический идентификатор узла в шестнадцатеричном формате. В этом примере две реплики объекта O (например, файла с текстом книги), O_1 и O_2, вставляются в разные узлы (O_1 → узел 4228 и O_2 → узел *AA*93). Идентификаторы O_1 и O_2 равны (4378 в шестнадцатеричном виде), т. к. O_1 и $O2$ – реплики одного и того же объекта O. Когда O_1 вставляется в его серверный узел (4228), отображение (4378, 4228) маршрутизируется из узла 4228 в узел 4377 (корневой узел идентификатора O_1). По мере приближения сообщения к корневому узлу идентификаторы

объекта и узла становятся все более похожи. Кроме того, отображение (4378, 4228) сохраняется во всех узлах на пути следования сообщения. Вставка O_2 производится аналогично. На рис. 9.13b показано, что если узел E791 ищет реплику O, то маршрут ассоциированного сообщения обрывается в узле 4361. Таким образом, приложения могут реплицировать данные в нескольких серверных узлах, полагаясь на то, что Tapestry будет направлять запросы к близлежащим репликам.

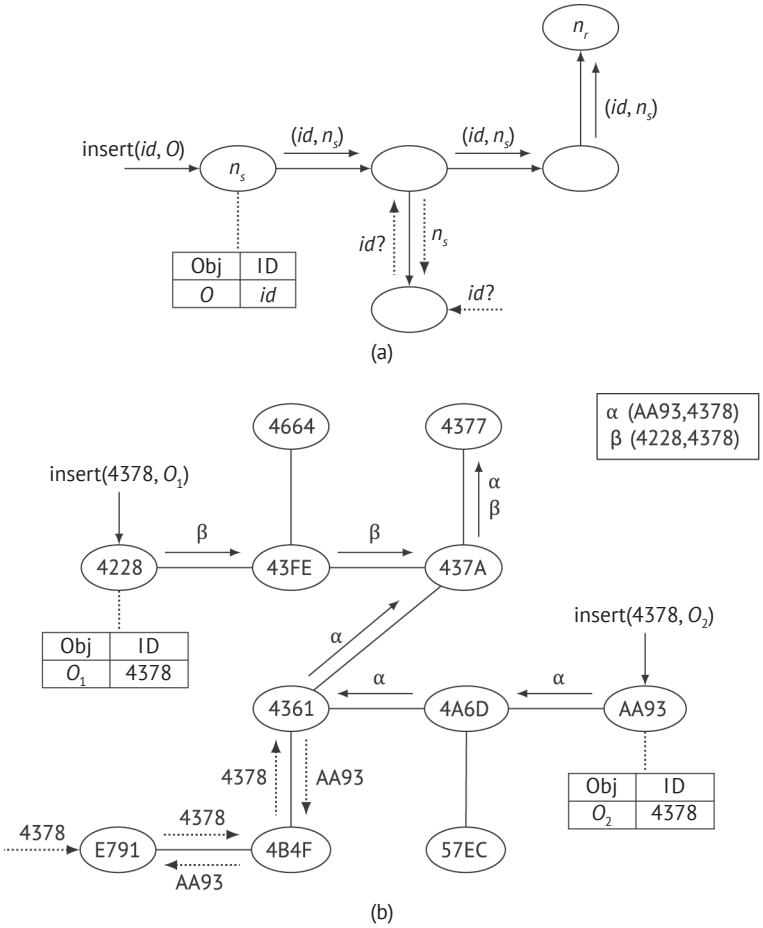


Рис. 9.13 ❖ Tapestry:
(a) публикация объекта; (b) управление репликами

9.4.2. Актуальность данных в DHT

Хотя DHT предлагает базовую поддержку репликации, взаимная согласованность реплик после обновления может быть нарушена из-за того, что узлы покидают сеть, или из-за конкурентных обновлений. Проиллюстрируем эту проблему на примере простого сценария обновления в типичной DHT.

Пример 9.7. Предположим, что в результате операции put(k, d_0) (выполняемой некоторыми узлами) два узла p_1 и p_2 получают и сохраняют данные d_0. Далее рассмотрим обновление (выполненное тем же или другим узлом) с помощью операции put(k, d_1), которая также отображает ключ k на узлы p_1 и p_2. Если p_2 недоступен (например, потому что покинул сеть), то обновляется только узел p_1. Когда p_2 снова присоединится к сети, реплики окажутся не согласованы: в p_1 хранится акутальное состояние данных, ассоциированных с k, а в p_2 – устаревшее.

Конкурентные обновления тоже ведут к проблемам. Рассмотрим два обновления put(k, d_2) и put(k, d_3) (выполненных разными узлами), которые доходят до p_1 и p_2 в противоположном порядке, так что последнее состояние p_1 равно d_2, а последнее состояние p_2 равно d_3. Тогда последующая операция get(k) вернет либо устаревшее, либо актуальное состояние в зависимости от того, к какому узлу обратилась, и нет никакого способа сказать, актуально состояние или нет. ♦

Для некоторых приложений (например, управление повесткой мероприятий, доски объявлений, управление кооперативными аукционами, управление бронированием), которые могли бы воспользоваться DHT, получение актуальных данных очень важно. Для поддержки актуальности данных в реплицированных DHT необходимо, чтобы всегда возвращалась актуальная реплика вне зависимости от того, покидали узлы сеть или нет, и от наличия конкурентных обновлений. Разумеется, согласованность реплик – более общая проблема, которая обсуждалась в главе 6, но в P2P-системах она особенно важна и трудна вследствие динамичного поведения узлов, присоединяющихся к системе и покидающих ее.

Было предложено решение, которое обеспечивает сразу и доступность, и актуальность данных. Для обеспечения высокой доступности данные в DHT реплицируются с помощью набора независимых хеш-функций H_r, которые называются *хеш-функциями репликации*. Узел, который в данный момент времени отвечает за ключ k относительно хеш-функции h, обозначается $rsp(k, h)$. Чтобы извлечь текущую реплику, каждая пара (k, *data*) снабжается временной меткой, и для каждой функции $h \in H_r$ пара (k, *newData*) реплицируется в узле $rsp(k, h)$, где *newData* = {*data*, *timestamp*}, т. е. *newData* состоит из исходных данных и временной метки. Получив запрос на данные, ассоциированные с ключом, мы можем вернуть любую из реплик, снабженных самой поздней временной меткой. Количество хеш-функций репликации, т. е. H_r, может быть разным в различных DHT. Например, если доступность узлов DHT низкая, то для повышения доступности можно увеличить размер H_r (например, до 30).

Это решение легло в основу *службы управления обновлениями* (Update Management Service – UMS), которая отвечает за эффективную вставку и выборку текущих реплик на основе механизма временных меток. Эксперименты показали, что накладные расходы на UMS с точки зрения стоимости коммуникации очень малы. После получения реплики UMS определяет, является ли она текущей, без сравнения с другими репликами и возвращает ее в качестве результата. Поэтому UMS не нужно получать все реплики, чтобы найти текущую, достаточно лишь службы поиска в DHT с операциями put и get.

Для генерирования временных меток UMS пользуется распределенной *службой временных меток на основе ключей* (Key-based Timestamping Service – KTS). Основная операция KTS, называемая `gen_ts`(*k*), по ключу *k* генерирует вещественное число, используемое как временная метка для *k*. Временные метки, генерируемые KTS, *монотонны*, т. е. если ts_i и ts_j – две метки, сгенерированные для одного и того же ключа в момент t_i и t_j соответственно, то $ts_j > ts_i$, если t_j позже t_i. Это позволяет упорядочить временные метки одного и того ж ключа по времени их генерирования. В KTS имеется также операция *last_ts*(*k*), которая возвращает последнюю временную метку, сгенерированную для ключа *k*. В любой момент времени `gen_ts`(*k*) генерирует не более одной временной метки для *k*, причем эти метки монотонно возрастают. Таким образом, если несколько узлов одновременно пытаются вставить в DHT пару (*k*, *data*), преуспеет только тот, который получил самую позднюю временную метку.

9.4.3. Урегулирование реплик

Урегулирование реплик – шаг вперед по сравнению с поддержанием актуальности данных, т. к. гарантирует взаимную согласованность реплик. Поскольку типичная P2P-сеть очень динамична (узлы присоединяются и уходят по собственному желанию), то решения на основе энергичной репликации не годятся, и необходимо прибегнуть к ленивой репликации (см. главу 6). В этом разделе мы опишем методы урегулирования, применяемые в системах OceanStore, P-Grid и APPA, чтобы продемонстрировать весь спектр возможных решений.

9.4.3.1. OceanStore

OceanStore – система управления данными, предназначенная для предоставления непрерывного доступа к хранимой на постоянных носителях информации. Она опирается на Tapestry и предполагает, что инфраструктура состоит из мощных не заслуживающих доверия серверов, соединенных высокоскоростными линиями связи. Из соображений безопасности для защиты данных применяется резервирование и криптографические методы. Для повышения производительности данные разрешено кешировать в любом месте сети.

OceanStore разрешает конкурентное обновление реплицированных объектов и для согласования реплик полагается на механизм урегулирования. Реплицированный объект может иметь несколько первичных и вторичных реплик в разных узлах. Все первичные реплики связаны ссылками и взаимодействуют между собой для достижения взаимной согласованности путем упорядочения обновлений. Для вторичных реплик допускается меньшая степень согласованности во имя повышения производительности и доступности. Таким образом, вторичные реплики могут быть не вполне актуальны, и их может быть больше, чем первичных. Вторичные реплики взаимодействуют между собой и с первичными репликами, применяя алгоритм распространения эпидемии.

На рис. 9.14 показано, как производятся обновления в OceanStore. Здесь R – единственный реплицируемый объект, а R и R_{sec} обозначают соответственно первичную и вторичную копии R. Все четыре узла, в которых хранится первичная копия, связаны между собой (на рисунке не показано). Пунктирными линиями представлены связи между узлами, хранящими первичные или вторичные реплики. Узлы n_1 и n_2 одновременно обновляют R. Управление такими обновлениями осуществляется следующим образом. Множество узлов, хранящих первичные копии R, *называется главной группой* R, они отвечают за упорядочение обновлений. Таким образом, n_1 и n_2 производят пробное обновление своих локальных вторичных копий и отправляют эти обновления главной группе R, а также другим случайно выбранным вторичным репликам (рис. 9.14а). Пробные обновления упорядочиваются главной группой по временным меткам, присвоенным узлами n_1 и n_2; одновременно эти обновления распространяются между вторичными репликами с по-

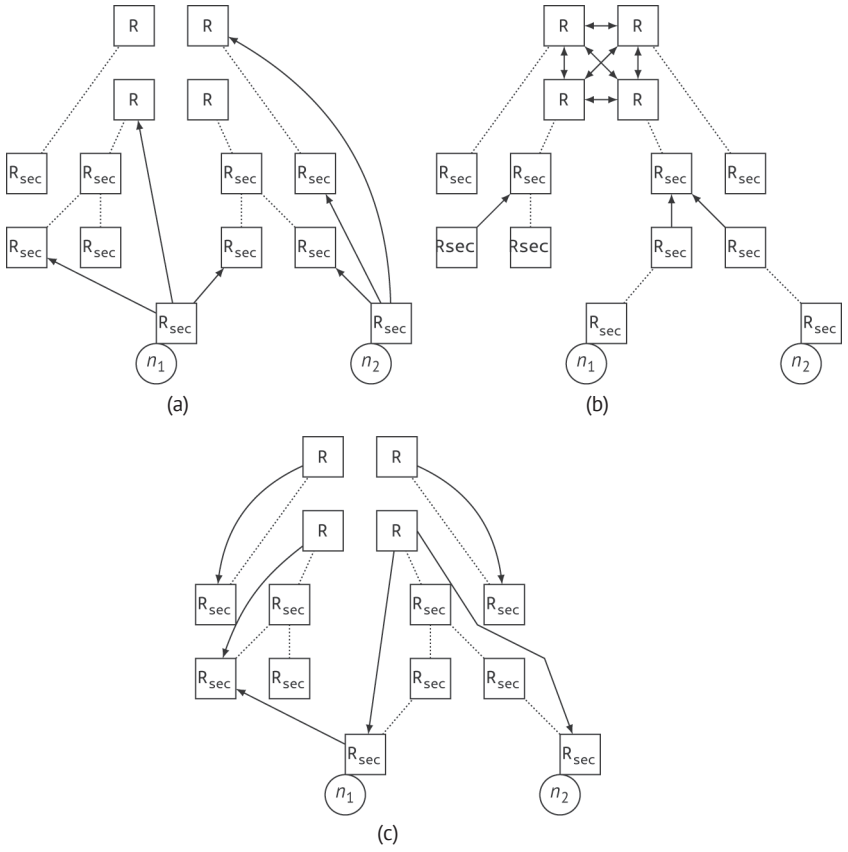


Рис. 9.14 ❖ Урегулирование в системе OceanStore:
(a) узлы n_1 и n_2 отправляют обновления главной группе R
и нескольким случайно выбранным вторичным репликам; (b) главная группа R
упорядочивает обновления, а вторичные реплики в это время распространяют их
с помощью алгоритма эпидемии; (c) после достижения согласия в главной группе
результат обновления рассылается всем вторичным репликам

мощью алгоритма эпидемии (рис. 9.14b). Как только главная группа придет к согласию, вторичным репликам рассылается результат обновления (рис. 9.14c), содержащий пробные[1] и зафиксированные данные.

9.4.3.2. P-Grid

P-Grid – это структурированная P2P-сеть со структурой двоичного дерева. В результате децентрализованного и самоорганизующегося процесса в P-Grid строится инфраструктура маршрутизации, которая адаптируется к имеющемуся распределению ключей, хранящихся в узлах. Этот процесс обеспечивает равномерную нагрузку на систему хранения данных и равномерную репликацию данных для поддержки доступности.

Чтобы решить проблему обновления реплицированных объектов, в P-Grid применяется алгоритм сплетен, не гарантирующий сильной согласованности. В P-Grid предполагается квазисогласованность реплик (вместо полной согласованности, которую трудно обеспечить в динамической среде).

Схема распространения обновлений состоит из двух этапов: проталкивания и вытягивания. Получив новое обновление реплицированного объекта `R`, узел *p* проталкивает его подмножеству узлов, владеющих репликами `R`, которые, в свою очередь, распространяют его другим узлам, владеющим репликами `R`, и т. д. Узлы, которые покинули сеть, а затем снова вошли в нее, узлы, которые долго не получали обновлений, и узлы, получившие запрос вытягивания, но не уверенные, что хранят последнее обновление, переходят к этапу вытягивания для урегулирования ситуации. На этом этапе несколько узлов связываются друг с другом, и самый актуальный из них выбирается поставщиком объекта.

9.4.3.3. APPA

Система APPA предлагает общее ленивое решение для распределенной репликации, которое гарантирует согласованность реплик в конечном счете. Для этого используется каркас действие–ограничение IceCube, который улавливает семантику приложений и разрешает конфликты при обновлении.

Семантика приложения описывается посредством ограничений между действиями обновления. Действие определяется программистом и представляет зависящую от приложения операцию (например, запись в файл или в документ либо транзакцию базы данных). Ограничение – это формальное представление инварианта приложения. Например, ограничение *predSucc*(a_1, a_2) устанавливает причинно-следственный порядок между действиями (действие a_2 выполняется только после успешного выполнения a_1), а ограничение *mutuallyExclusive*(a_1, a_2) говорит, что можно выполнить либо a_1, либо a_2, но не оба сразу. Цель урегулирования – взять набор действий и ассоциированных с ними ограничений и построить *расписание*, т. е. список упорядоченных действий, не нарушающих ограничений. Чтобы уменьшить сложность этой

[1] Пробными называются данные, которые первичные реплики еще не зафиксировали.

процедуры, множество подлежащих упорядочению действий разбивается на подмножества, называемые *кластерами*. Кластер – это подмножество действий с ограничениями, которое можно упорядочить независимо от других кластеров. Затем *глобальное расписание* строится путем конкатенации упорядоченных действий в кластерах.

Данные, управляемые алгоритмом урегулирования APPA, хранятся в структурах, называемых *объектами урегулирования*. У каждого объекта урегулирования имеется уникальный идентификатор, позволяющий хранить его в DHT. Репликация данных производится следующим образом. Сначала узлы выполняют локальные действия для обновления реплики объекта с соблюдением определенных пользователем ограничений. Затем эти действия (и ассоциированные с ними ограничения) сохраняются в DHT под идентификатором объекта. Наконец, узлы-примирители извлекают действия и ограничения из DHT и строят глобальное расписание, урегулируя конфликтующие действия на основе семантики приложения. Это расписание локально выполняется в каждом узле, чем гарантируется согласованность в конечном счете. Любой подключенный к сети узел может попытаться начать урегулирование, пригласив другие доступные узлы поучаствовать в нем. В каждый момент времени может выполняться только одна процедура урегулирования. Урегулирование действий обновления состоит из шести описанных ниже распределенных шагов. Узлы на шаге 2 начинают урегулирование. Результаты каждого шага становятся входными данными для следующего:

- **шаг 1 – выбор узлов:** подмножество подключенных к сети, хранящих реплики, выбирается на роль примирителей, исходя из стоимости коммуникации;
- **шаг 2 – группировка действий:** примирители читают действия из журналов действий и помещают действия, которые пытались обновить одни и те же объекты, в одну группу, поскольку они потенциально конфликтуют. Группы действий, пытающихся обновить объект R, хранятся в объекте урегулирования «журнал действий R» (L_R);
- **шаг 3 – создание кластера:** примирители читают группы действий из журналов действий и разбивают их на кластеры семантически зависимых конфликтующих действий: два действия a_1 и a_2 семантически независимы, если приложение считает безопасным выполнять их вместе в любом порядке, даже если они обновляют один и тот же объект; в противном случае a_1 и a_2 семантически зависимы. Кластеры, построенные на этом шаге, хранятся в объекте урегулирования «множество кластеров»;
- **шаг 4 – расширение кластеров:** при создании кластеров заданные пользователем ограничения не принимаются во внимание. На этом шаге примирители расширяют кластеры, добавляя в них новые конфликтующие действия согласно пользовательским ограничениям;
- **шаг 5 – объединение кластеров:** после расширения кластеры могут пересекаться (т. е. у двух кластеров имеются общие группы действий). На этом шаге примирители объединяют пересекающиеся кластеры. После этого кластеры становятся независимыми, т. е. не существует ограничений, в которых участвовали бы действия из разных кластеров;

- **шаг 6 – упорядочение кластеров:** на этом шаге примирители упорядочивают действия в каждом кластере. Упорядоченные действия, ассоциированные с каждым кластером, хранятся в объекте урегулирования «расписание». Конкатенация упорядоченных действий из всех кластеров дает глобальное расписание, выполняемое всеми узлами-репликами.

На каждом шаге алгоритм урегулирования старается воспользоваться параллелизмом данных, т. е. несколько узлов одновременно выполняют независимые операции над различными подмножествами данных (например, упорядочение различных кластеров).

9.5. Блокчейн

Ставшая популярной на волне увлечения биткойном и другими криптовалютами, технология блокчейн представляет собой недавно появившуюся P2P-инфраструктуру, в которой можно эффективно и безопасно хранить транзакции, совершенные двумя сторонами. Эта тема стала предметом рекламной шумихи и споров. С одной стороны, имеются рьяные сторонники, например Ито, Нарула и Али, в 2017 году заявившие на весь мир, что блокчейн – прорывная технология, которая «сотворит с финансовой системой то же, что интернет сотворил со СМИ». С другой стороны, имеются непримиримые противники, например знаменитый экономист Н. Рубини, в 2018 году назвавший блокчейн самой «переоцененной и бесполезной технологией в истории человечества». Как всегда, истина находится где-то посередине.

Блокчейн был придуман для биткойна, чтобы решить присущую прежним цифровым валютам проблему двойного расходования, не вводя центральный доверенный орган. 3 января 2009 года Сатоси Накамото[1] создал первый исходный блок с уникальной транзакцией на 50 биткойнов на свое имя. С тех пор появилось много других блокчейнов, например Ethereum в 2013 году и Ripple в 2014 году. Технологии сопутствовал громкий успех, и криптовалюты активно использовались для перевода денежных средств и высокорисковых инвестиций, например первичного размещения криптомонет (ICO) как альтернативы первичному публичному размещению акций (IPO). Перечислим потенциальные преимущества криптовалют на основе блокчейна:

- низкая плата за транзакцию (задается отправителем для ускорения обработки), не зависящая от переведенной суммы;
- пониженный риск для торговцев (невозможно мошенническое оспаривание сделки);
- безопасность и контроль (например, защита от кражи личности);
- доверие к самому блокчейну без какого-либо центрального органа.

---

[1] Псевдоним человека или группы лиц, разработавших биткойн. Было много споров о том, кто скрывается за этим псевдонимом.

Однако криптовалюты также часто использовались для афер и незаконной деятельности (покупок в темном интернете, отмывания денег, кражи и т. д.), что повлекло за собой предупреждения от административных органов и стало причиной регулирования в некоторых странах. Существуют и другие проблемы:

- неустойчивость: поскольку не существует поддержки со стороны государства или центрального банка (в отличие от таких сильных валют, как доллар и евро);
- отсутствие связи с реальной экономикой, что является питательной почвой для спекуляций;
- высокая волатильность, когда курс обмена на реальную валюту (устанавливаемый на криптовалютных биржах) может резко меняться на протяжении всего нескольких часов;
- высокая вероятность надувания криптовалютных пузырей, как было в 2017 году.

Таким образом, у криптовалют на основе блокчейна есть плюсы и минусы. Но вовсе необязательно ограничиваться только криптовалютами, у блокчейна есть и много других полезных применений. Изначально блокчейн представляет собой открытый распределенный реестр, в который можно безопасно записывать на постоянное хранение транзакции, доступные ряду компьютеров. Это сложная распределенная инфраструктура базы данных, в которой сочетается несколько технологий: P2P, репликация данных, протокол достижения консенсуса и шифрование с открытым ключом. Термином Блокчейн 2.0 обозначают приложения, которые можно запрограммировать с помощью блокчейна, выйдя за рамки финансовых сделок и разрешив обмен активами без облеченных властью посредников. Примерами могут служить смарт-контракты, постоянные цифровые идентификаторы, права интеллектуальной собственности, ведение блогов, голосование, проверка деловой репутации и т. д.

9.5.1. Определение блокчейна

Для регистрации финансовых сделок между двумя сторонами традиционно использовался посредник в виде централизованного реестра, т. е. база данных всех сделок, управляемая доверенным полномочным органом, например клиринговой палатой. В цифровом мире у такого централизованного подхода есть несколько проблем. Во-первых, он становится точкой общего отказа и привлекательной мишенью для хакеров. Во-вторых, он подразумевает сосредоточение участников, например крупных финансовых учреждений. В-третьих, заключение сложных сделок с несколькими посредниками, которые обычно пользуются разнородными системами и правилами, сталкивается с трудностями и требует много времени.

Блокчейн по существу представляет собой распределенный реестр, разделяемый узлами, участвующими в P2P-сети. Он организован в виде реплицированной базы блоков, в которую можно только дописывать. Блоки – это цифровые контейнеры транзакций, защищенные шифрованием с открытым

ключом. Код каждого нового блока основан на предшествующем блоке, что исключает любые манипуляции. Блокчейн виден всем участникам, которые поддерживают копии базы данных в режиме с несколькими главными узлами (см. главу 6) и сотрудничают в рамках протокола достижения консенсуса для проверки транзакций в блоках. После проверки и записи в блок транзакцию нельзя ни изменить, ни удалить, что защищает блокчейн от манипуляций. Узлы-участники могут не полностью доверять друг другу, некоторые могут даже вести себя злонамеренно (по-византийски), т. е. предъявлять разные значения различным узлам-наблюдателям. Таким образом, в общем случае публичный блокчейн, в частности тот, что используется в биткойне, должен быть устойчивым к византийским отказам.

Заметим, что цель типичной одноранговой структуры данных типа DHT – обеспечить быстрый и масштабируемый поиск. У блокчейна же цель совсем другая – управлять непрерывно растущим списком блоков безопасным и защищенным от манипуляций способом. Но масштабируемость целью не является, поскольку блокчейн не распределен между узлами P2P-сети.

По сравнению с централизованным реестром блокчейн обладает следующими преимуществами:

- большее доверие к сделкам и обмену активами, поскольку доверием пользуются сами данные, а не участники;
- повышенная надежность (отсутствие точки общего отказа) благодаря репликации;
- встроенная безопасность, обеспеченная организацией цепочки блоков и применением шифрования с открытым ключом;
- эффективные и дешевые сделки между участниками, особенно если сравнивать с длинной цепочкой посредников.

Блокчейны можно использовать на рынках двух видов: открытые, например криптовалюта или открытые аукционы, когда участником может стать любой желающий, и закрытые, например управление цепочками поставок или здравоохранение, когда участники известны. Таким образом, существует важное различие между открытыми и закрытыми (или эксклюзивными) блокчейнами.

Открытый блокчейн (например, биткойн) – это открытая, не требующая разрешения для доступа сеть, возможно, очень большого масштаба. Участники неизвестны, не заслуживают доверия и могут входить в сеть и выходить из нее без уведомления. Обычно у каждого участника есть псевдоним, который позволяет проследить всю историю его сделок, а иногда даже идентифицировать.

Закрытый блокчейн – это закрытая эксклюзивная сеть, обычно гораздо меньшего масштаба, чем открытый блокчейн. Гарантируется, что проверять транзакции могут только идентифицированные участники, получившие разрешение. Доступ к блокчейну можно разрешить лишь авторизованным участникам, что повышает степень защищенности данных. Базовая структура открытого и закрытого блокчейнов одинакова, а основное различие между ними в том, кому (частному лицу, группе лиц или компании) разрешено участвовать в сети и кто управляет сетью.

9.5.2. Инфраструктура блокчейна

В этом разделе мы познакомимся с инфраструктурой блокчейна, первоначально предложенной для биткойна, с акцентом на обработку транзакций. Узлы-участники называются *полными узлами*, в отличие от других, например облегченных клиентских узлов, которые обрабатывают цифровые бумажники. Когда новый полный узел присоединяется к сети, он синхронизируется с известными узлами с помощью системы доменных имен (DNS) для получения копии блокчейна. После этого он сможет создавать транзакции и стать «майнером», т. е. участвовать в проверке блоков – этот процесс называется «майнингом».

Обработка транзакции состоит из трех основных шагов:

1) создание транзакции, после того как два пользователя пришли к согласию относительно обмена информацией о транзакции: адреса бумажников, открытые ключи и т. д.;
2) группировка транзакций в блок и связывание его с предыдущим блоком;
3) проверка блока (и транзакций) с использованием «майнинга», добавление проверенного блока в блокчейн и репликация в сети.

Далее в этом разделе мы рассмотрим эти шаги более детально.

9.5.2.1. Создание транзакции

Рассмотрим биткойновскую транзакцию между владельцем и получателем монеты. Транзакция защищена шифрованием с открытым ключом и цифровой подписью. У каждого владельца имеется открытый и закрытый ключи. Для подписания транзакции владелец монеты:

- создает хеш-свертку комбинации предыдущей транзакции (с помощью которой он получает монеты) с открытым ключом предыдущего владельца;
- подписывает свертку своим закрытым ключом.

Эта подпись добавляется в конец транзакции, в результате чего создается цепочка транзакций, соединяющая всех владельцев (см. рис. 9.15). Затем владелец монеты публикует транзакцию в сети, рассылая ее всем остальным узлам. Зная открытый ключ владельца монеты, создавшего транзакцию, любой узел в сети может проверить подпись транзакции.

9.5.2.2. Группировка транзакций в блоки

Двойное расходование – потенциальный дефект любой схемы цифровых наличных, заключающийся в том, что один цифровой денежный знак можно потратить несколько раз. В отличие от физических денег, цифровой денежный знак представляет собой файл, который можно скопировать или подделать.

Каждый узел-майнер (содержащий копию блокчейна) получает публикуемые транзакции, проверяет их и группирует в блоки. Для того чтобы принять транзакцию и включить ее в блок, майнер следует определенным правилам, в частности проверяет, что входные данные корректны и что монета не по-

трачена более одного раза в результате атаки (см. описание атаки типа 51 % ниже). Может оказаться, что злонамеренный майнер попытается принять транзакцию, нарушающую некоторые правила, и включить ее в блок. В таком случае блок не получит консенсусного одобрения других майнеров, соблюдающих правила, а значит, не будет принят и включен в блокчейн. Таким образом, если большинство майнеров соблюдают правила, то система работает. Как показано на рис. 9.16, каждый новый блок строится на основе предыдущего блока в цепочке, для чего вычисляется хеш-свертка (H-значение) адреса предыдущего блока, что защищает блок от любых манипуляций. В текущей версии биткойна размер блока равен 1 мегабайту, что является компромиссом между эффективностью и безопасностью.

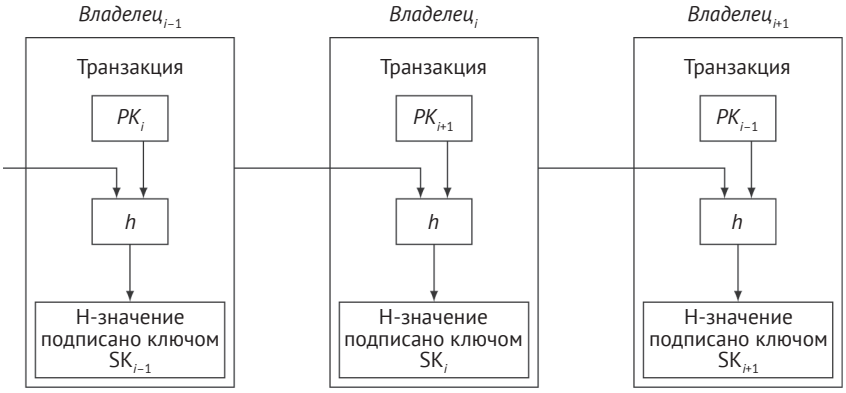


Рис. 9.15 ❖ Связывание транзакций в цепочку

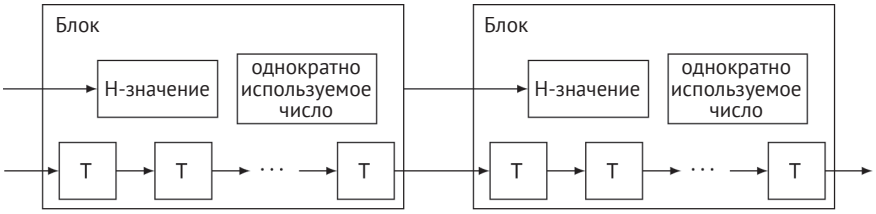


Рис. 9.16 ❖ Связывание блоков в цепочку

Может возникнуть проблема случайного или намеренного ветвления. Поскольку разные блоки параллельно проверяются разными узлами, один узел может в любой момент видеть несколько цепочек-кандидатов. Например, на рис. 9.17 узел может увидеть блоки 7a и 6b, причем оба происходят от блока 5. Для решения проблемы предложено правило самой длинной цепочки, требующее, чтобы выбирался блок, находящийся в самой длинной цепочке. В примере на рис. 9.17 для построения следующего блока 7b будет выбран блок 7a. Смысл этого правила в том, чтобы уменьшить количество транзакций, подлежащих повторной передаче. Например, транзакции в блоке 6b должны быть повторно переданы клиентом (который увидит, что блок

не прошел проверку). Пока что транзакции в проверяемом блоке прошли только предварительную выборку и ожидают подтверждения. Любой новый блок, принятый в цепочку после проверки транзакции, считается подтверждением. В технологии биткойна транзакция считается достоверной после 6 подтверждений (в среднем для этого требуется примерно час). На рис. 9.17 более достоверные транзакции обозначаются более темным цветом (блок 6b светлее, потому что его транзакции не будут подтверждены).

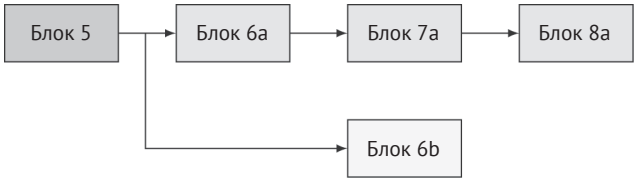


Рис. 9.17 ❖ Правило самой длинной цепочки

Ветвление может быть не только случайным, но и намеренным, что полезно, когда в код блокчейна добавляются новые возможности (изменение протокола) или требуется отменить последствия атаки либо катастрофических ошибок. Ветвления бывают двух типов: *мягкие* и *жесткие*. Мягкое ветвление поддерживает обратную совместимость: старые версии программы считают блоки, созданные по новым правилам, допустимыми. Однако это облегчает задачу атакующим. Знаменитый случай применения жесткого ветвления – атака на блокчейн Ethereum в 2016 году против смарт-контракта на предоставление венчурного капитала. Ethereum разветвился, но без единодушной поддержки со стороны сообщества, сопровождающего программное обеспечение. В результате образовалось два блокчейна: новый Ethereum и старый Ethereum Classic. Заметим, что столкновение имело больше философскую и этическую, нежели техническую природу.

9.5.2.3. Консенсусная проверка блока

Поскольку блоки создаются параллельно разными узлами, для их проверки и добавления в блокчейн необходим консенсус. Заметим, что в общем случае открытого блокчейна с неизвестными участниками традиционные протоколы достижения консенсуса типа Paxos (см. раздел 5.4.5) неприменимы. Протокол консенсуса в блокчейне биткойна основан на *майнинге*[1]. Вкратце протокол достижения консенсуса выглядит следующим образом:

1) узлы-майнеры конкурируют (как в лотерее) за порождение новых блоков. Потребляя очень много вычислительной мощности, каждый майнер старается создать однократно используемое число *nonce* (number used once) для блока (см. рис. 9.16);

[1] Майнинг буквально означает добычу (полезных ископаемых) и употребляется по аналогии с добычей золота, т. е. процессом изготовления монет, который является частью протокола.

2) получив *nonce*, майнер добавляет блок в блокчейн и рассылает всем узлам сети;
3) другие узлы верифицируют новый блок, проверяя *nonce* (это просто);
4) поскольку сразу много узлов пытаются первыми добавить блок в блокчейн, основанная на лотерее система вознаграждения случайным образом выбирает один из конкурирующих блоков и выплачивает победителю награду, которая на сегодняшний день составляет 12,5 биткойна (сначала было 50). В результате запас монет увеличивается.

Майнинг задуман как трудная задача. Чем больше майнинговых мощностей в сети, тем труднее вычислить *nonce*. Это позволяет контролировать включение новых блоков в систему («инфляцию») – в среднем один блок каждые 10 минут. Трудность майнинга основана на доказательстве выполнения работы (Proof of Work – PoW), для чего требуется произвести некоторое длительное вычисление, результат которого легко проверяется. Впервые принцип PoW был применен для защиты от DoS-атак. В блокчейне для биткойнов используется алгоритм Hashcash, основанный на хеш-функции SHA-256. Цель – найти такое значение v, что $h(f(block, v)) < T$, где

1) h – хеш-функция SHA-256;
2) f – функция, которая объединяет v с информацией в блоке, чтобы *nonce* нельзя было вычислить заранее;
3) T – значение, общее для всех узлов и отражающее размер сети;
4) v – 256-разрядное число, начинающееся n нулевыми битами.

В среднем затраты на получение доказательства выполнения работы экспоненциально зависят от количества требуемых нулевых битов, т. е. вероятность успеха мала и приблизительно равна $1/2^n$. Это дает преимущество мощным узлам, которые теперь представляют собой большие кластеры GPU. Однако проверить результат очень просто, для этого нужно лишь вычислить одну хеш-функцию.

Потенциальная проблема майнинга, основанного на принципе PoW, – *атака типа 51 %*, которая позволяет атакующему аннулировать законные транзакции и дважды израсходовать фонды. Для этого атакующий (майнер или коалиция майнеров) должен располагать более чем 50 % общей вычислительной мощности, доступной для майнинга. Тогда можно будет изменить полученную цепочку (например, удалить транзакцию) и создать более длинную цепочку, которая будет выбрана большинством по правилу самой длинной цепочки.

9.5.3. Блокчейн 2.0

Блокчейн первого поколения, ознаменовавшийся появлением биткойнов, позволяет регистрировать транзакции и обмениваться криптовалютой без облеченных властью посредников. Блокчейн 2.0 – значительное развитие этой парадигмы, которая вышла за пределы финансовых транзакций и распространилась на обмен любыми видами активов. Путь проторила система Ethereum, сделавшая блокчейн программируемым и предоставившая разработчикам возможность создавать API службы поверх блокчейна.

Приложения обладают следующими важнейшими характеристиками: обмен активами (посредством транзакций), наличие нескольких участников, возможно неизвестных друг другу, и безусловное доверие к данным. У технологии блокчейн 2.0 есть многочисленные применения в разных отраслях, например: финансовые службы и микроплатежи, цифровые права, управление цепочками поставок, хранение записей о здоровье пациентов, интернет вещей, маркировка продуктов питания. Большую часть этих приложений можно поддержать с помощью закрытых блокчейнов. В таком случае основные преимущества – повышенная конфиденциальность и управляемость и более эффективная проверка транзакций, поскольку участникам можно доверять и нет необходимости предъявлять доказательство выполнения работы.

Важная функция, поддерживаемая блокчейном 2.0, – смарт-контракты. Так называется самоисполняемый контракт, в код которого включены условия и положения контракта. Пример простого смарт-контракта – контракт на обслуживание между двумя сторонами, одна из которых запрашивает услугу и готова заплатить за нее оговоренную цену, а вторая оказывает услугу и после ее выполнения получает платеж. В блокчейне контракты могут находиться в полностью или частично выполненном состоянии без участия человека и включать много участников, например устройств интернета вещей. Основное преимущество хранения смарт-контрактов в блокчейне заключается в том, что код, реализующий контракт, виден всем и может быть верифицирован. Но после помещения в блокчейн контракт уже нельзя изменить. С технической точки зрения, главная проблема – написать код без ошибок, для чего правильнее всего было бы воспользоваться верификацией кода. Важная коллективная инициатива – создание блокчейнов и соответствующих инструментов с открытым исходным кодом в рамках проекта Hyperledger, спонсируемого фондом Linux Foundation. Этот проект был основан в 2015 году компаниями IBM, Intel, Cisco и другими. Перечислим его основные компоненты:

- Hyperledger Fabric (IBM, цифровой актив): инфраструктура эксклюзивного блокчейна, поддерживающая смарт-контракты, конфигурируемый протокол достижения консенсуса и службы членства;
- Sawtooth (Intel): новый механизм консенсуса, «доказательство затраченного времени», надстроенный над средой доверенного выполнения;
- Hyperledger Iroha (Soramitsu): основан на Hyperledger Fabric с акцентом на мобильные приложения.

9.5.4. Проблемы

Блокчейн часто рекламируют как прорывную технологию для регистрации транзакций и проверки записей, делая упор на финансовой отрасли. В частности, наличие средств для программирования приложений и бизнес-логики на базе блокчейна открывает для разработчиков много возможностей, например смарт-контракты. Некоторые сторонники, в т. ч. шифропанки, даже считают ее силой, потенциально способной установить истинную демократию и равенство, когда физические лица и малый бизнес смогут конкурировать с огромными корпорациями.

Однако существуют важные ограничения, особенно в случае открытого блокчейна – наиболее общей инфраструктуры.

- Сложность и масштабируемость, а особенно трудности эволюции правил работы, изменение которых требует ветвления блокчейна.
- Постоянно увеличивающийся размер цепочки и высокое энергопотребление (для доказательства выполнения работы).
- Возможность атаки типа 51 %.
- Невысокая конфиденциальность, поскольку пользователи только скрываются за псевдонимами. Например, совершение транзакции с некоторым пользователем может раскрыть все остальные его транзакции.
- Непредсказуемая длительность совершения транзакций – от нескольких минут до нескольких дней.
- Отсутствие контроля и регулирования, что затрудняет отслеживание и налогообложение сделок государством.
- Проблемы безопасности: если закрытый ключ потерян или украден, нет никакого обходного пути.

Существует несколько направлений исследований в области распределенных систем, программной инженерии и управления данными, призванных снять эти ограничения.

- Масштабируемость и безопасность открытого блокчейна. Этот вопрос вызвал новый всплеск интереса к протоколам достижения консенсуса, которые были бы эффективнее доказательства выполнения работы: доказательство доли владения, доказательство обладания, доказательство использования, доказательство времени владения. Существуют, впрочем, и другие узкие места, помимо консенсуса. Но главной проблемой остается компромисс между производительностью и безопасностью. Bitcoin-NG – блокчейн нового поколения с двумя типами блоков: ключевые блоки, включающие PoW, ссылку на предыдущий блок и вознаграждение за майнинг, делающее вычисление PoW более эффективным, и микроблоки, которые включают транзакции, но не требуют PoW.
- Управление смарт-контрактами, включая сертификацию и верификацию кода, эволюцию контракта (распространение изменений), оптимизацию и контроль выполнения.
- Блокчейн и управление данными. Поскольку блокчейн – это просто распределенная база данных, его можно улучшить, позаимствовав идеи проектирования систем баз данных. Например, наличие декларативного языка могло бы упростить определение, верификацию и оптимизацию сложных смарт-контрактов. BigchainDB – новая СУБД, в которой идеи распределенных баз данных, в частности развитая модель транзакций, контроль доступа на основе ролей и запросы, применены к поддержке масштабируемого блокчейна. Для понимания того, где возникают узкие места, необходимо также тестирование производительности. BLOCKBENCH – каркас, созданный для тестирования производительности закрытых блокчейнов при различных рабочих нагрузках.
- Интероперабельность блокчейнов. Существует много блокчейнов, каждый со своими протоколами и API. Для определения стандартов и продвижения межблокчейновых транзакций создана организация Blockchain Interoperability Alliance (BIA).

9.6. Заключение

Благодаря распределению хранения и обработки данных между автономными одноранговыми узлами сети P2P-системы могут масштабироваться, не нуждаясь в мощных серверах. В настоящее время такие приложения для совместного использования данных, как BitTorrent, eDonkey или Gnutella, каждодневно используются миллионами людей. Технология P2P успешно применялась для масштабирования управления данными в облаке, например в хранилище ключей и значений DynamoDB (см. раздел 11.2.1). Однако в части функциональности базы данных эти приложения по-прежнему играют ограниченную роль.

Продвинутые P2P-приложения, например для коллективного потребления (скажем, каршеринг), должны уметь работать с семантически обогащенными данными (XML или RDF-документами, реляционными таблицами и т. д.). Для поддержки таких приложений нужно в значительной мере пересмотреть методы работы с распределенными базами данных (управление схемой, контроль доступа, обработку запросов, управление транзакциями, управление согласованностью, надежность и репликацию). Что касается управления данными, то основные требования к P2P-системе – автономность, выразительная способность языка запросов, эффективность, качество обслуживания и отказоустойчивость. В какой мере эти требования достижимы, зависит от архитектуры P2P-сети (неструктурированная, структурированная на основе DHT или супероднораноговая). Неструктурированные сети характеризуются лучшей отказоустойчивостью, но могут быть крайне неэффективны, поскольку для маршрутизации запросов применяется лавинная передача. У гибридных сетей больше потенциал для удовлетворения высокоуровневых требований к управлению данными. Однако DHT-системы лучше приспособлены для поиска по ключу и могут комбинироваться с супероднораноговыми сетями для выполнения более сложных видов поиска.

Поначалу большая часть работ по разделению данных в P2P-системах была посвящена управлению схемой и обработке запросов, в особенности работе с семантически обогащенными данными. Но с появлением технологии блокчейн акцент сместился на управление обновлениями, репликацию, транзакции и контроль доступа при работе со сравнительно простыми данными. Методы P2P также вызвали интерес в связи с вертикальным масштабированием управления данными в контексте грид-вычислений и в связи с обеспечением конфиденциальности данных в контексте информационного поиска и аналитики.

Возрождение интереса к управлению данными в P2P-системах наблюдается в двух основных контекстах: блокчейн и граничные вычисления. В области блокчейна исследования, которые мы подробно обсуждали в конце раздела 9.5, сосредоточены на масштабируемости и безопасности открытого блокчейна (например, протоколы достижения консенсуса), управлении смарт-контрактами, в частности использовании декларативных языков, тестировании производительности и интероперабельности блокчейнов. Что касается граничных вычислений, обычно с помощью устройств интернета вещей, то мобильные граничные серверы можно было бы организовать

в P2P-сеть для разгрузки при решении задач управления данными. Тогда возникают новые задачи на пересечении мобильных и P2P-вычислений.

9.7. Библиографические замечания

Управление данными в «современных» P2P-системах характеризуется высокой степенью распределенности, внутренне присущей гетерогенностью и непостоянством состава сети. Эта тема подробно обсуждается в нескольких книгах, в т. ч. [Vu et al. 2009, Pacitti et al. 2012]. Краткий обзор содержится в работе [Ulusoy 2007]. Обсуждение требований, архитектур и проблем, с которыми сталкиваются P2P-системы управления данными, см. в работах [Bernstein et al. 2002, Daswani et al. 2003, Valduriez and Pacitti 2004]. Несколько P2P-систем такого рода описано в работе [Aberer 2003].

В неструктурированных P2P-сетях проблема лавинной передачи данных решается одним из двух методов. Выбор подмножества узлов, которым направлять запрос, предложен в работе [Kalogeraki et al. 2002]. Использование случайного блуждания для выбора множества соседей описано в работе [Lv et al. 2002], применение индекса соседства – в работе [Yang and Garcia-Molina 2002], а ведение индекса ресурсов для определения списка соседей, которые с большой вероятностью расположены в направлении искомого узла, – в работе [Crespo and Garcia-Molina 2002]. Альтернативное предложение – использовать протокол распространения эпидемии – обсуждается в работе [Kermarrec and van Steen 2007], которая основана на протоколе сплетен, описанном в работе [Demers et al. 1987]. Подходы к масштабированию протокола сплетен рассмотрены в работе [Voulgaris et al. 2003].

Структурированные P2P-сети обсуждаются в работе [Ritter 2001, Ratnasamy et al. 2001, Stoica et al. 2001]. Как и DHT, динамическое хеширование успешно применялось для решения проблем масштабируемости очень больших распределенных файловых структур [Devine 1993, Litwin et al. 1993]. Основанные на DHT наложенные сети можно классифицировать по геометрии и алгоритму маршрутизации [Gummadi et al. 2003]. Мы более-менее подробно разобрали следующие DHT: Tapestry [Zhao et al. 2004], CAN [Ratnasamy et al. 2001] и Chord [Stoica et al. 2003]. Упомянутые нами структурированные P2P-сети рассматриваются в следующих источниках: PHT [Ramabhadran et al. 2004], P-Grid [Aberer 2001, Aberer et al. 2003a], BATON [Jagadish et al. 2005], BATON* [Jagadish et al. 2006], VBI-tree [Jagadish et al. 2005], P-дерево [Crainiceanu et al. 2004], SkipNet [Harvey et al. 2003] и Skip Graph [Aspnes and Shah 2003]. В работе [Schmidt and Parashar 2004] описана система, в которой для определения структуры используются кривые, заполняющие пространство, а в работе [Ganesan et al. 2004] предложена система на основе гиперпрямоугольников.

Примерами супероднораноговых сетей являются Edutella [Nejdl et al. 2003] и JXTA.

Хорошее обсуждение вопросов отображения схем в P2P-системах можно найти в работе [Tatarinov et al. 2003]. Попарное отображение схем используется в системах Piazza [Tatarinov et al. 2003], LRM [Bernstein et al. 2002], Hyperion [Kementsietsidis et al. 2003] и P-Grid [Aberer et al. 2003b]. Отображе-

ние на основе методов машинного обучения используется в системе GLUE [Doan et al. 2003b], отображение на основе общего согласия – в системах APPA [Akbarinia et al. 2006, Akbarinia and Martins 2007] и AutoMed [McBrien and Poulovassilis 2003], отображение с применением информационного поиска – в системах PeerDB [Ooi et al. 2003] и Edutella [Nejdl et al. 2003]. Семантическое переформулирование запроса с помощью попарного отображения схем в социальных P2P-сетях рассматривается в работе [Bonifati et al. 2014].

Подробный обзор обработки запросов в P2P-системах содержится в работе [Akbarinia et al. 2007b], которая легла в основу разделов 9.2 и 9.3. Важный вид запросов в P2P-системах – запросы типа «первые k». Обзор методов обработки таких запросов в реляционных системах баз данных см. в работе [Ilyas et al. 2008]. В частности, эффективен пороговый алгоритм (TA), независимо предложенный несколькими исследователями [Nepal and Ramakrishna 1999, Güntzer et al. 2000, Fagin et al. 2003]. TA лег в основу нескольких алгоритмов в P2P-системах, в частности DHT [Akbarinia et al. 2007a]. Более эффективный по сравнению с TA алгоритм наилучшей позиции предложен в работе [Akbarinia et al. 2007c]. Несколько TA-подобных алгоритмов было предложено для распределенной обработки запросов типа «первые k», например TPUT [Cao and Wang 2004].

Обработке запросов типа «первые k» в P2P-системах было уделено много внимания: в неструктурированных системах – PlanetP [Cuenca-Acuna et al. 2003] и APPA [Akbarinia et al. 2006]; в системах на основе DHT – APPA [Akbarinia et al. 2007a]; в суперодноранговых системах – Edutella [Balke et al. 2005]. Решения задачи об обработке запросов с соединением в P2P-системах реализованы в системе PIER [Huebsch et al. 2003]. Для обработки запросов по диапазону применялось локально-чувствительное хеширование [Gupta et al. 2003], другие решения предложены в системах PHT [Ramabhadran et al. 2004] и BATON [Jagadish et al. 2005].

Обзор репликации в P2P-системах, представленный в работе [Martins et al. 2006b], лег в основу изложения в разделе 9.4. Полное решение проблемы актуальности данных в реплицированных DHT, т. е. способность находить самую актуальную реплику, описано в работе [Akbarinia et al. 2007d]. Задача об урегулировании реплик решается в системах OceanStore [Kubiatowicz et al. 2000], P-Grid [Aberer et al. 2003a] и APPA [Martins et al. 2006a, Martins and Pacitti 2006, Martins et al. 2008]. Каркас действие–ограничение предложен для системы IceCube [Kermarrec et al. 2001].

Методы P2P также применялись для вертикального масштабирования управления данными в контексте grid-вычислений [Pacitti et al. 2007], граничных и мобильных вычислений [Tang et al. 2019] и для обеспечения конфиденциальности данных в аналитике [Allard et al. 2015].

Блокчейн – сравнительно недавняя технология, вызывающая много споров. У нее имеются как горячие сторонники [Ito et al. 2017], так и авторитетные противники, например известный экономист Н. Рубини [Roubini 2018]. Основные идеи были изложены в первопроходческой работе по блокчейну для реализации биткойна [Nakamoto 2008]. С тех пор было предложено много других блокчейнов для криптовалют, например Ethereum и Ripple. Первые реализации были в основном созданы разработчиками не из академических

кругов. Поэтому главный источник информации – веб-сайты, технические описания и блоги. Академические исследования блокчейна начались недавно. В 2016 году начал издаваться академический журнал Ledger – первый из посвященных различным аспектам технологии блокчейн (в информатике, технике, праве, экономике и философии). В сообществе ученых, занимающихся распределенными системами, основное внимание уделялось улучшению безопасности и производительности протоколов, для чего, например, была разработана технология Bitcoin-NG [Eyal et al. 2016]. В сообществе, связанном с управлением данными, распространяются пособия, представленные на крупных конференциях, например [Maiyya et al. 2018], обзорные статьи, например [Dinh et al. 2018], и проекты конкретных систем, например BigchainDB. Для расшивки узких мест необходимо также тестирование производительности, например с помощью системы BLOCKBENCH [Dinh et al. 2018].

Упражнения

Задача 9.1. В чем заключается фундаментальное различие между одноранговой и клиент-серверной архитектурой? Верно ли, что P2P-система с централизованным индексом эквивалентна клиент-серверной системе? Назовите основные преимущества и недостатки файлообменных P2P-систем с точки зрения:

- конечных пользователей;
- владельцев файлов;
- сетевых администраторов.

Задача 9.2 (**). Наложенная P2P-сеть построена поверх физической сети, обычно интернета. Их топологии различны, и два узла, являющиеся соседями в P2P-сети, могут далеко отстоять друг от друга в физической сети. Каковы преимущества и недостатки такого разбиения на уровни? Какое влияние такая организация оказывает на дизайн трех основных типов P2P-сетей (неструктурированных, структурированных и супероднoранговых)?

Задача 9.3 (*). Рассмотрим неструктурированную P2P-сеть на рис. 9.4, и пусть расположенный в левом нижнем углу узел отправляет запрос на поиск ресурса. Проиллюстрируйте и обсудите следующие две стратегии поиска с точки зрения полноты результатов:

- лавинное распространение с TTL=3;
- протокол сплетен, когда каждый узел обладает частичным представлением не более чем о 3 соседях.

Задача 9.4 (*). Рассмотрим рис. 9.7, обратив особое внимание на структурированные сети. Воспользовавшись шкалой оценок от 1 до 5 (вместо низкая, умеренная, высокая), уточните сравнение для трех основных типов DHT: дерево, гиперкуб и кольцо.

Задача 9.5 (**). Пусть наша цель – спроектировать социальное приложение с P2P-сетью поверх DHT. Приложение должно предоставлять основные функции социальных сетей: регистрация профиля нового пользователя, при-

глашение и поиск друзей, создание списков друзей, отправка сообщения друзьям, чтение сообщений от друзей, комментирование сообщений. Предположим, что каждый пользователь является узлом DHT и что DHT предоставляет общие операции `put` и `get`.

Задача 9.6 (**). Предложите P2P-архитектуру социальной сети, в которой распределенные по сети сущности представлены парами (ключ, данные). Опишите реализацию следующих операций: создать или удалить пользователя, создать или удалить друга, читать сообщения от списка друзей. Обсудите плюсы и минусы своего проекта.

Задача 9.7 (**). Тот же вопрос, но с дополнительным требованием: частные данные (например, профиль пользователя) должны храниться в узле пользователя.

Задача 9.8. Обсудите сходство и различие отображения схем в системах мультибаз данных и в P2P-системах. В частности, сравните подход «локальная как представление», описанный в главе 7, с попарным отображением схем, описанным в разделе 9.2.1.

Задача 9.9 (*). Алгоритм FD обработки запросов типа «первые k» в неструктурированных P2P-сетях (см. алгоритм 9.4) основан на лавинном распространении. Предложите вариант FD, в котором вместо лавинного распространения используется случайное блуждание или протокол сплетен. В чем его плюсы и минусы?

Задача 9.10 (*). Примените алгоритм TPUT (алгоритм 9.2) с $k = 3$ ко всем трем спискам базы данных на рис. 9.10. Для каждого шага алгоритма покажите промежуточные результаты.

Задача 9.11 (*). Тот же вопрос применительно к алгоритму DHTop (алгоритм 9.5).

Задача 9.12 (*). В алгоритме 9.6 предполагается, что соединяемые входные отношения размещены в произвольных местах DHT. В предположении, что одно из отношений уже хешировано по атрибутам соединения, предложите усовершенствование алгоритма 9.6.

Задача 9.13 (*). Для повышения доступности данных в DHT существует общее решение: реплицировать пары (*ключ*, *данные*) в нескольких узлах, используя несколько хеш-функций. При этом возникает проблема, показанная на рис. 9.7. Альтернатива – использовать нереплицированную DHT (с одной хеш-функцией) и обязать узлы реплицировать пары (*ключ*, *данные*) в некоторых своих соседях. Как это повлияет на сценарий в примере 9.7? Каковы плюсы и минусы этого подхода с точки зрения доступности и балансировки нагрузки?

Задача 9.14 (*). Обсудите сходства и различия открытого и закрытого (эксклюзивного) блокчейна. В частности, проанализируйте свойства, которыми должен обладать протокол проверки транзакций.

Глава 10

Обработка больших данных

В прошлом десятилетии мы стали свидетелями «информационно емких» или «информационно-центрических» приложений, в которых основой для решения задач является анализ больших объемов разнородных данных. Часто их называют *приложениями больших данных*, и ведутся исследования специальных систем, которые поддерживали бы управление такими данными и их обработку, – *систем обработки больших данных*. Нужда в подобных приложениях возникает во многих областях: здравоохранении, социальных сетях, изучении окружающей среды и других. Большие данные – основная часть науки о данных, объединяющей такие разные дисциплины, как управление данными, статистический анализ данных, машинное обучение и прочие. Цель этой науки – извлечение новых знаний из данных. Чем больше данных, тем лучше результаты, получаемые наукой о данных, но этому сопутствуют проблемы, связанные с управлением и обработкой.

Не существует точного определения приложений или систем больших данных, но обычно они характеризуются «четырьмя V» (хотя некоторые включают и дополнительные V: value (ценность), validity (достоверность) и т. д.).

1. **Volume (объем).** Наборы данных, используемые в таких приложениях, очень велики, как правило, порядка петабайтов (1 ПБ = 10^{15} байт), а с ростом интернета вещей скоро достигнут зеттабайтов (1 ЗБ = 10^{21} байт). Для сравнения скажем, что, согласно отчетам Google, в 2016 году для хранения данных, загружаемых пользователями YouTube, *каждый день* дополнительно требовалось 1 ПБ. Ожидается экспоненциальный рост этой величины с 10-кратным увеличением каждые пять лет (так что к моменту, когда вы читаете эту книгу, ежедневный рост, вероятно, уже составляет 10 ПБ). В Facebook хранится 250 млрд изображений (по состоянию на 2018 год). Компания Alibaba сообщала, что в периоды пиковой нагрузки в 2017 году за шесть часов в результате действий покупателей генерировались журналы общим размером 320 ПБ.
2. **Variety (разнородность).** Традиционные (обычно под этим понимают реляционные) СУБД предназначены для работы с хорошо структурированными данными, описываемыми схемой. Для приложений больших данных это уже не так, и приходится иметь дело с разнородными данными. Помимо структурированной информации, они могут включать

также изображения, текст, аудио и видео. Согласно некоторым отчетам, 90 % генерируемых ныне данных не структурированы. Системы больших данных должны органично управлять такими данными и обрабатывать их.

3. **Velocity (скорость возникновения).** Важный аспект приложений больших данных заключается в том, что иногда данные поступают с высокой скоростью, а приложения должны успевать их обрабатывать. Например, Facebook должен обрабатывать 900 млн ежедневно загружаемых фотографий; Alibaba сообщала, что в пиковые периоды необходимо было обрабатывать 470 млн событий журнала каждую секунду. Порядок этих чисел настолько велик, что у системы нет возможности предварительно сохранить данные, их необходимо обрабатывать в режиме реального времени.
4. **Veracity (достоверность).** Большие данные поступают в приложение из разных источников, не все из которых надежны или заслуживают доверия – данные могут быть зашумлены, статистически смещены, разные копии могут быть не согласованы между собой, возможна также сознательная дезинформация. Обычно такие данные называются «грязными», и это неизбежно, поскольку количество источников растет вместе с объемом данных. Сообщалось, что затраты на обработку грязных данных ежегодно возрастают на 3 млрд долларов только в экономике США. Системы больших данных должны «очищать» эти данные и отслеживать их происхождение, чтобы можно было делать какие-то выводы о том, насколько они достойны доверия. Еще один важный аспект достоверности – «правдивость» данных, т. е. отсутствие шума, смещения и сознательных манипуляций. Данные должны заслуживать доверия – это критически важно.

Эти характеристики сильно отличаются от тех, с которыми имеют дело традиционные СУБД (до сих пор именно они были в фокусе нашего внимания), поэтому требуются новые системы, методологии и подходы. Можно, наверное, возразить, что параллельные СУБД (см. главу 8) хорошо справляются с серьезными объемами, потому что известны примеры, когда они обрабатывали очень большие наборы данных; однако системы, способные решать все описанные выше задачи, заслуживают внимания. Все это темы активных исследований и разработок, а наша цель в этой и следующей главах – описать, какие подходы к системной инфраструктуре рассматриваются для решения первых трех задач; достоверность не зависит от этого обсуждения, ее можно считать отдельной темой, которой мы далее не будем касаться, но в библиографических замечаниях дадим ссылки на некоторые работы в этом направлении. Напомним, что мы кратко обсуждали этот вопрос в главе 7 (точнее, в разделе 7.1.5), и еще вернемся к нему в контексте управления вебданными в главе 12 (конкретно в разделе 12.6.3).

По сравнению с традиционными СУБД, в системах управления большими данными используется другой программный стек с уровнями, показанными на рис. 10.1. Управление большими данными опирается на уровень распределенного хранения, который обычно обеспечивает хранение данных в файлах или объектах, распределенных по узлам кластера без разделения ресурсов.

К данным, находящимся в распределенных файлах, имеется прямой доступ со стороны каркаса обработки данных, который позволяет программисту выражать код параллельной обработки без вмешательства СУБД. Поверх каркаса обработки данных могут быть построены средства написания скриптов и декларативных запросов (наподобие SQL). Для управления разнородными данными на уровне доступа к данным обычно развертываются NoSQL-системы, но можно использовать системы потоковой обработки и даже поисковые системы. Наконец, на самом верху находятся различные инструменты для более сложной аналитической обработки больших данных, в т. ч. средства машинного обучения (МО). Этот программный стек, воплощением которого является, например, обсуждаемая ниже система Hadoop, благоприятствует интеграции слабо связанных (как правило, с открытым исходным кодом) компонентов. Так, СУБД типа NoSQL, как правило, поддерживает различные системы хранения (например, HDFS, но не только). Такие системы часто развертываются в публичной или частной облачной среде. В этой и следующей главах обсуждение будет строиться в соответствии с описанной многоуровневой архитектурой.

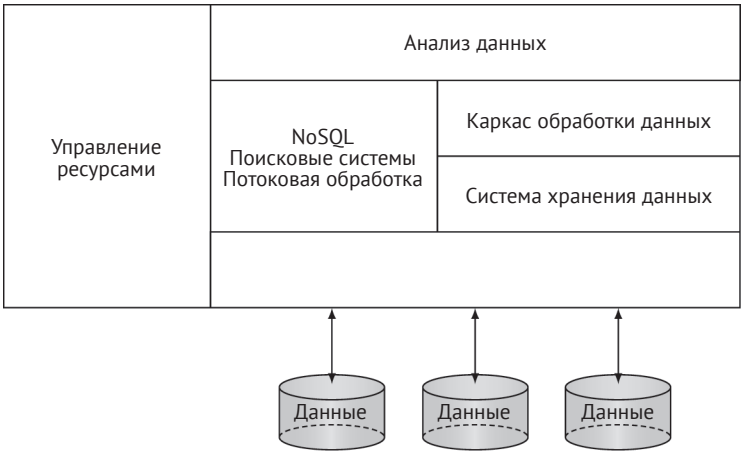


Рис. 10.1 ❖ Программный стек для управления большими данными

Далее в этой главе мы будем рассматривать компоненты данной архитектуры снизу вверх и попутно остановимся на двух из четырех V, характеризующих системы больших данных. Раздел 10.1 посвящен распределенным системам хранения. В разделе 10.2 рассматриваются два важных каркаса обработки данных: MapReduce и Spark. Вместе с разделом 10.1 этот раздел охватывает вопросы масштабируемости, т. е. первое V – «volume» (объем). В разделе 10.3 мы обсудим потоковую обработку данных, т. е. третье V – «velocity» (скорость возникновения). В разделе 10.4 рассматриваются графовые системы, ориентированные на анализ графовых данных; это второе V – «variety» (разнородность). Вопросы разнородности затрагиваются также в разделе 10.5, где обсуждается недавно возникшая область – озера данных. В озерах

накапливаются данные из многих источников – как структурированные, так и неструктурированные. Часть этой архитектуры, связанную с технологиями NoSQL, мы отложим до следующей главы.

10.1. Распределенные системы хранения

Управление большими данными опирается на уровень распределенного хранения, где данные обычно хранятся в файлах или объектах, распределенных по узлам кластера без разделения ресурсов. Это одно из важных отличий от современных СУБД, где данные хранятся в блоках. Интересно проследить, как менялась эта часть программного стека со временем. Самые первые СУБД, основанные на иерархической или сетевой модели, строились как расширения файловой системы, как, например, COBOL с его ссылками между файлами. Первые реляционные СУБД тоже строились поверх файловой системы. Например, знаменитая СУБД INGRES была реализована поверх файловой системы Unix. Но доступ к данным с помощью средств файловой системы общего назначения был крайне неэффективен, поскольку СУБД не могла управлять ни кластеризацией данных на диске, ни кешем в оперативной памяти. Критика подхода на основе файлов в основном была направлена на отсутствие в операционной системе средств управления данными (в то время). Поэтому архитектура реляционных СУБД эволюционировала в сторону блочной, с использованием низкоуровневого интерфейса с диском, предоставляемого операционной системой. Блочный интерфейс обеспечивает прямой эффективный доступ к дисковым блокам (единица выделения места на диске). Сегодня все реляционные СУБД являются блочными и, следовательно, полностью контролируют управление диском. В параллельных СУБД сохранен тот же подход, в основном для того, чтобы облегчить переход от централизованных систем. Главная причина возврата к использованию файловой системы заключается в том, чтобы сделать распределенную систему хранения отказоустойчивой и масштабируемой и тем самым упростить построение следующих уровней управления данными.

В этом контексте уровень распределенного хранения обычно предлагает два решения для хранения данных в узлах кластера – в виде объектов или файлов. Эти два решения взаимно дополняют друг друга, поскольку преследуют разные цели и могут сочетаться.

Объектное хранилище управляет данными как объектами. Объект содержит собственно данные вместе с переменным количеством метаданных, а также уникальный идентификатор (oid), выбираемый из плоского пространства имен. Таким образом, объект можно представить в виде тройки ⟨oid, данные, метаданные⟩, и после создания к нему можно обращаться непосредственно по `oid`. Поскольку данные и метаданные упакованы в один объект, объекты легко перемещать между узлами распределенной системы хранения. В отличие от файловых систем, где тип метаданных для всех файлов одинаков, объекты могут содержать разное количество метаданных. Это позволяет гибко описывать, как объекты защищены, как их можно репли-

цировать, когда можно удалять и т. д. Использование плоского пространства открывает возможность для управления огромным количеством (миллиарды или триллионы) неструктурированных объектов данных. Наконец, к объектам можно обращаться с помощью REST API, включающего команды `put` и `get`, которые легко использовать в протоколах интернета. Объектные хранилища особенно полезны для хранения очень большого числа сравнительно малых объектов: фотографий, вложений в электронные письма и т. д. Поэтому такой подход пользуется популярностью у облачных поставщиков, которые обслуживают подобные приложения.

Файловое хранилище управляет данными, находящимися в неструктурированных файлах (последовательностях байтов), поверх которых может быть реализована структура в виде записей фиксированной или переменной длины. Файловая система организует файлы в виде иерархии каталогов и для каждого файла, но отдельно от него хранит метаданные (имя файла, положение в каталоге, владелец, длина содержимого, время создания, время последнего обновления, права доступа и т. п.). Таким образом, чтобы получить доступ к содержимому файла, нужно сначала прочитать его метаданные. Из-за этого файловое хранилище больше подходит для совместного доступа к файлам локально в центре обработки данных, когда количество файлов ограничено (порядка сотен тысяч). Для работы с большими файлами, содержащими очень много записей, необходимо разбить файл на части и распределить их между узлами кластера с помощью распределенной файловой системы. Одна из самых авторитетных распределенных файловых систем – Google File System (GFS). Далее в этом разделе мы опишем GFS, а также обсудим сочетание систем объектного и файлового хранения, которое обычно встречается в облаке.

10.1.1. Google File System

Файловая система GFS была разработана компанией Google для внутреннего использования и применяется во многих приложениях и системах Google, в частности Bigtable.

Как и другие распределенные файловые системы, GFS нацелена на обеспечение производительности, масштабируемости, отказоустойчивости и доступности. Однако системы, на которых она развертывается, – кластеры без разделения ресурсов – вызывают трудности, т. к. состоят из большого числа (порядка тысяч) серверов на базе недорого оборудования. Поэтому вероятность отказа сервера в любой момент времени высока, из-за чего трудно добиться отказоустойчивости. В GFS эта проблема решается с помощью репликации и механизма отработки отказов, как мы вскоре увидим. Она также оптимизирована под такие информационно емкие приложения Google, как поисковая система и аналитика данных. Эти приложения отличаются следующими особенностями. Во-первых, файлы очень велики, обычно несколько гигабайтов, и содержат много объектов, например веб-документов. Во-вторых, рабочая нагрузка включает много операций чтения и дописывания в конец, тогда как произвольные обновления редки. Читаются как

большие фрагменты данных (порядка 1 МБ), так и небольшие куски (порядка нескольких КБ). Операции дописывания обычно велики, и в один файл одновременно может дописывать много клиентов. В-третьих, поскольку рабочая нагрузка состоит из большого числа операций чтения и дописывания, высокая пропускная способность важнее низкой задержки.

GFS организует файлы в виде дерева каталогов и идентифицирует их по путевым именам. Предоставляется интерфейс к файловой системе с традиционными операциями (`create`, `open`, `read`, `write`, `close` и `delete`) и двумя дополнительными операциями: `snapshot`, которая создает копию файла или дерева каталогов, и `record append`, которая позволяет эффективно дописывать данные (записи, англ. record) в конец файла конкурентным клиентам.

Запись дописывается атомарно, т. е. в виде непрерывной строки байтов, начиная с позиции, определенной GFS. Это позволяет избежать распределенного управления блокировками, которое было бы необходимо в случае традиционной операции записи (которую в принципе можно было бы использовать для дописывания данных в конец файла).

Архитектура GFS изображена на рис. 10.2. Файлы разбиваются на *порции* фиксированной длины 64 МБ. В узлах кластера находятся клиенты GFS, которые предоставляют интерфейс GFS приложениям, серверы порций, которые отвечают за хранение порций, и один мастер-узел GFS, на котором хранятся метаданные файлов, в т. ч. пространство имен, информация для контроля доступа и о месте нахождения файлов. Мастер присваивает каждой порции уникальный идентификатор в момент создания и для надежности реплицирует ее по крайней мере на трех серверах порций. Для доступа к данным порции клиент должен сначала узнать у мастера ее местоположение, а затем запросить данные у одной из реплик.

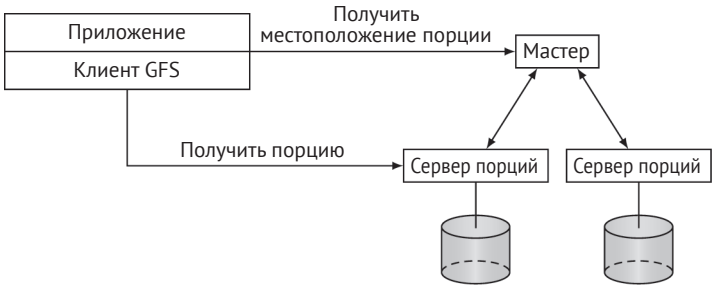


Рис. 10.2 ❖ Архитектура GFS

Такая архитектура с единственным мастером проста, а поскольку мастер используется в основном для поиска порций и не хранит данные порций, то он не является узким местом. Кроме того, ни на клиентах, ни на серверах порций данные не кешируются, т. к. это не ускорило бы объемные операции чтения. Еще одно упрощение – ослабленная модель согласованности для конкурентных операций записи и дописывания в конец. Это означает, что приложения должны помнить об ослабленной согласованности и бороться с ней с помощью таких методов, как создание контрольных точек и записей

со встроенным самоконтролем. Наконец, чтобы система сохраняла высокую доступность в условиях частых отказов узлов, GFS использует репликацию и автоматическую отработку отказа. Каждая порция реплицируется на нескольких серверах (по умолчанию GFS хранит три реплики). Мастер-узел периодически посылает каждому серверу порций контрольные сообщения. В случае отказа сервера порций мастер автоматически выполняет процедуру отработки отказа, перенаправляя все запросы на доступ к файлам работающему серверу с репликой. GFS также реплицирует все данные мастера на теневом мастере, который автоматически принимает на себя управление в случае выхода мастера из строя.

Существуют реализации GFS с открытым исходным кодом, например распределенная файловая система Hadoop (Hadoop Distributed File System – HDFS), которую мы обсудим в разделе 10.2.1. Есть и другие распределенные файловые системы для кластерных архитектур с открытым исходным кодом, например GlusterFS для кластера без разделения ресурсов и Global File System 2 (GFS2) для кластера с общим диском – обе разрабатываются компанией Red Hat для Linux.

10.1.2. Сочетание объектного и файлового хранения

Наметилась важная тенденция к сочетанию хранения объектов и файлов в одной системе, чтобы поддержать одновременно большое число объектов и большие файлы. Впервые это было осуществлено в системе Ceph – программной платформе с открытым исходным кодом, которая разработана компанией Red Hat для экзабайтного кластера без разделения ресурсов. В Ceph операции с данными и метаданными разделены, для чего таблицы размещения файлов исключены и заменены функциями распределения данных, спроектированными для гетерогенных динамичных кластеров с ненадежными устройствами хранения объектов (object storage device – OSD). Это позволяет Ceph задействовать встроенную в OSD логику для распределения по разным уровням сложности, присущей доступу к данным, сериализии обновлений, репликации, обеспечению надежности, обнаружению отказов и восстановлению. В настоящее время Ceph и GlusterFS – две основные платформы хранения, предлагаемые Red Hat для кластеров без разделения ресурсов.

С другой стороны, HDFS стала стандартом де-факто масштабируемого и надежного управления файловыми системами для больших данных. Таким образом, имеется большой соблазн добавить средства хранения объектов в HDFS, чтобы упростить хранение данных поставщикам и пользователям облаков. В системе Azure HDInsight – основанном на Hadoop решении Microsoft для управления большими данными в облаке – HDFS интегрирована с Azure Blob Storage, диспетчером хранения объектов, и может непосредственно работать со структурированными и неструктурированными данными. В контейнерах Blob Storage данные хранятся в виде пар ключ-значение, иерархии каталогов не существует.

10.2. Каркасы для обработки больших данных

Существует важный класс приложений больших данных, которым необходимо управлять данными без накладных расходов на управление полноценной базой данных. Кроме того, облачным службам нужна масштабируемость для приложений, которые легко разбить на большое число небольших параллельно исполняемых задач, – так называемых естественно распараллеливаемых приложений. Для таких случаев, когда масштабируемость важнее декларативных средств запросов, поддержки транзакций и согласованности базы данных, предложена платформа параллельной обработки MapReduce. Основная идея – упростить параллельную обработку с помощью платформы распределенных вычислений, предлагающей два интерфейса: map (отображение) и reduce (редукция). Программисты самостоятельно реализуют функции отображения и редукции, а система отвечает за планирование и синхронизацию соответствующих задач. Эта архитектура была оптимизирована в системе Spark, но большая часть приведенного ниже обсуждения относится к обоим каркасам. Начнем с обсуждения базового каркаса MapReduce (раздел 10.2.1), а затем перейдем к оптимизациям в Spark (раздел 10.2.2).

Обычно называют следующие преимущества каркасов обработки этого типа:

1) **гибкость**. Пользователь, который пишет код функций `map` и `reduce`, может очень гибко указать, как именно следует обрабатывать данные, а не ограничиваться средствами SQL. Программист может написать простые функции `map` и `reduce` для обработки больших объемов данных на многих машинах (или узлах, если употреблять терминологию, принятую в параллельных СУБД)[1], ничего не зная о том, как распараллелить обработку задания MapReduce;
2) **масштабируемость**. Во многих существующих приложениях главная проблема – масштабирование при возрастании количества данных. В частности, в облачных приложениях желательна *эластичная масштабируемость*, т. е. динамическое повышение и снижение потребления ресурсов в соответствии с изменяющимися требованиями. Такая модель обслуживания с «оплатой по факту» принята практически всеми поставщиками облачных служб, и MapReduce со своей параллельной обработкой данных может без труда поддержать ее;
3) **эффективность**. При работе с MapReduce необязательно загружать данные в базу, что позволяет избежать высоких затрат на внесение данных. Поэтому эта технология весьма эффективна для приложений, где обрабатывать данные нужно только один (или немного) раз;
4) **отказоустойчивость.** В MapReduce каждое задание разбивается на много мелких задач, которые раздаются разным машинам. Если произойдет сбой задачи или отказ машины, то задача передается другой

[1] В литературе по MapReduce их обычно называют *исполнителями* (worker), но мы используем термин *узел* и в главе о параллельных СУБД, и в следующей главе, посвященной NoSQL. Читатель должен понимать, что оба термина употребляются как синонимы.

машине, способной справиться с нагрузкой. Входные данные для задания хранятся в распределенной файловой системе, где для повышения доступности данные многократно реплицируются. Поэтому невыполненную задачу `map` можно корректно повторить, загрузив другую реплику. Невыполненную задачу `reduce` тоже можно повторить, повторно получив данные от завершенных задач `map`.

Критики MapReduce в основном недовольны сокращением функциональности, большим объемом программирования и непригодностью технологии для некоторых типов приложений (например, требующих итеративных вычислений). MapReduce не требует наличия схемы и не предлагает такого высокоуровневого языка программирования, как SQL. За вышеупомянутую гибкость приходится расплачиваться значительными затратами на (обычно трудоемкое) программирование силами пользователей. Поэтому работа, которую можно было бы выполнить с помощью сравнительно простых SQL-команд, может потребовать написания большого объема кода для MapReduce, и вовсе необязательно этот код будет допускать повторное использование. К тому же MapReduce не располагает встроенными средствами индексирования и оптимизации запросов, а всегда прибегает к последовательному просмотру (это можно считать преимуществом или недостатком, в зависимости от точки зрения).

10.2.1. Обработка данных в MapReduce

Как уже было сказано, MapReduce – это упрощенный подход к параллельной обработке данных, ориентированный на выполнение в компьютерном кластере. Он позволяет программисту выразить свои намерения в простом функциональном стиле и скрывает детали параллельной обработки, балансировки нагрузки и обеспечения отказоустойчивости. Модель программирования включает две определяемые пользователем функции, `map()` и `reduce()`, со следующей семантикой:

`map`	$(k_1, v_1) \rightarrow list(k_2, v_2)$
`reduce`	$(k_2, list(v_2)) \rightarrow list(v_3)$

Функция `map` применяется к каждой записи входного набора данных и вычисляет ноль или более промежуточных пар (ключ, значение). Функция `reduce` применяется ко всем значениям с одинаковым ключом и вычисляет результат их комбинирования. Поскольку эти функции работают с независимыми данными, их можно автоматически вычислять параллельно в разных секциях данных на нескольких узлах кластера.

На рис. 10.3 показана схема работы MapReduce в кластере. На вход функции `map` подается множество пар ключ-значение. Когда в систему вводится задание MapReduce, в вычислительных узлах запускаются задачи отображения (это процессы, которые называются *преобразователями*). Каждая такая задача применяет функцию `map` к каждой назначенной ей паре (k_1, v_1). Для одной пары ключ-значение может быть порождено ноль или более проме-

жуточных пар – список $list(k_2, v_2)$. Промежуточные результаты сохраняются в локальной файловой системе и сортируются по ключам. После того как все задачи отображения будут завершены, движок MapReduce уведомляет задачи редукции (это тоже процессы, но называются они *редукторами*) о том, что те могут начать работу. Редукторы параллельно читают файлы, созданные преобразователями, и производят их сортировку слиянием с целью объединить пары ключ-значение в множество новых пар $(k_2, list(v_2))$, так что все значения с одним и тем же ключом собираются в список и подаются на вход функции `reduce`; этот процесс, часто называемый *тасованием*, по существу является параллельной сортировкой. Функция `reduce` применяет к данным определенную пользователем логику. Результатом обычно является список значений, который записывается в систему хранения.

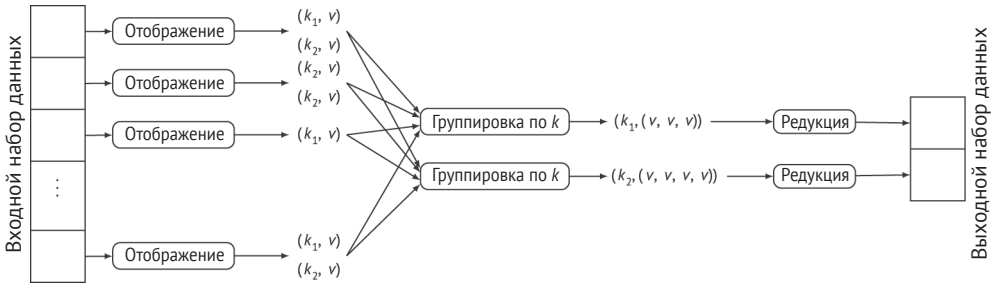


Рис. 10.3 ❖ Схема работы MapReduce

Помимо написания функций `map` и `reduce`, у программиста есть и другие средства управления процессом путем создания определенных пользователем функций (user-defined function – UDF) (например, форматов ввода-вывода и функций разбиения).

Пример 10.1. Рассмотрим отношение `EMP(ENO, ENAME, TITLE, CITY)` и следующий SQL-запрос, который для каждого города возвращает количество работников, чья фамилия содержит строку «Smith».

```
SELECT CITY, COUNT(*)
FROM   EMP
WHERE  ENAME LIKE "%Smith"
GROUP BY CITY
```

Для обработки этого запроса с помощью MapReduce можно написать такие функции Map и Reduce (мы приводим только их псевдокод).

```
Map (Input: (TID,EMP), Output: (CITY,1))
      if EMP.ENAME like "\%Smith" return (CITY,1)
Reduce (Input: (CITY,list(1)), Output: (CITY, SUM(list(1)))
      return (CITY,SUM(1))
```

Функция `map` параллельно применяется ко всем кортежам `EMP`. Она получает пару (TID, `EMP`), в которой ключом является идентификатор кортежа `EMP`

(TID), а значением – сам кортеж EMP, и, если кортеж удовлетворяет условию, возвращает одну пару (CITY, 1). Отметим, что разбирать кортеж и выделять атрибуты должна функция map. Затем все пары (CITY, 1) с одинаковым значением CITY группируются вместе, и для каждого города CITY создается пара (CITY, list(1)). После этого параллельно применяется функция reduce – она вычисляет количество вхождений каждого значения CITY и порождает результат запроса. ♦

10.2.1.1. Архитектура MapReduce

При обсуждении особенностей MapReduce мы остановимся только на одной конкретной реализации: Hadoop. Стек Hadoop показан на рис. 10.4, это реализация архитектуры обработки больших данных на рис. 10.1. В Hadoop для хранения данных используется распределенная файловая система Hadoop (HDFS), хотя возможно развертывание и с другими системами хранения. HDFS и движок Hadoop связаны слабо; они могут быть развернуты на одних и тех же или на разных узлах. В HDFS имеется два типа узлов: *узлы имен* и *узлы данных*. Узел имен хранит информацию о секционировании данных и следит за состоянием узлов данных. Данные, импортированные в HDFS, разбиваются на порции одинакового размера, и узел имен распределяет порции по разным узлам данных, которые отвечают за назначенные им порции. Кроме того, узел имен играет роль сервера словаря, т. е. предоставляет информацию приложениям, которые ищут конкретную порцию данных.

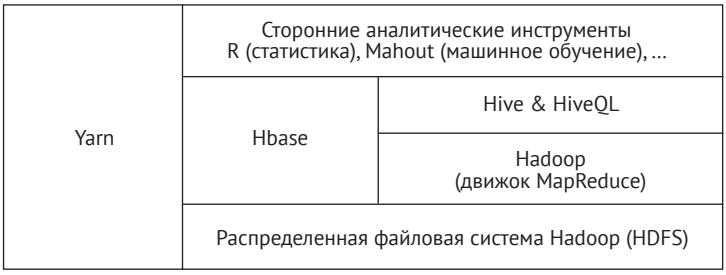


Рис. 10.4 ❖ Стек Hadoop

Отделение движка Hadoop от системы хранения дает возможность независимо увеличивать или уменьшать масштаб уровней обработки и хранения по мере необходимости. В разделе 10.1 мы обсуждали различные подходы к проектированию распределенной системы хранения и приводили примеры. Каждая порция данных, хранящаяся на одной машине кластера, является входом процесса-преобразователя. Поэтому если набор данных разбит на *k* порций, то Hadoop создаст *k* преобразователей для его обработки (или наоборот).

Движок Hadoop состоит из узлов двух типов: *мастер* и *исполнители* (рис. 10.5). Мастер управляет потоком выполнения задач в узлах-исполнителях, используя для этой цели модуль *планировщика*; в Hadoop он называ-

ется *отслеживатель заданий* (job tracker). Каждый исполнитель отвечает за задачу отображения или редукции. Базовая реализация движка MapReduce должна включать следующие модули, первые три из которых обязательны, а остальные являются расширениями:

1) *планировщик* отвечает за назначение задач отображения и редукции узлам-исполнителям с учетом близости данных, состояния сети и прочей статистики узлов. Он же управляет отказоустойчивостью, переназначая сбойный процесс другим узлам-исполнителям (если возможно). Дизайн *планировщика* существенно влияет на производительность всей системы MapReduce;
2) *модуль отображения* последовательно просматривает порцию данных и вызывает определенную пользователем функцию `map()` для обработки входных данных. Созданные промежуточные результаты (множество пар ключ-значение) группируются по ключу, кортежи в каждой группе сортируются, после чего мастеру посылается уведомление о том, где находятся результаты;
3) *модуль редукции* забирает данные у преобразователей, получив уведомление от мастера. После того как все промежуточные результаты получены, редуктор группирует данные с одинаковыми ключами. Затем в каждой группе применяется определенная пользователем функция, и результаты записываются в распределенное хранилище;
4) *модули ввода и вывода*. Модуль ввода отвечает за разбор данных в разных входных форматах и их разбиение на пары ключ-значение. Этот модуль дает движку возможность работать с разными источниками данных: текстовыми файлами, двоичными файлами и даже файлами базы данных. Модуль вывода определяет выходной формат преобразователей и редукторов;
5) *модуль комбинирования*. Цель этого модуля – уменьшить стоимость тасования путем выполнения локального процесса редукции для пар ключ-значение, созданных преобразователем;
6) *модуль разбиения* определяет способ тасования пар ключ-значение при передаче от преобразователей редукторам. По умолчанию применяется функция разбиения $f(key) = h(key) \% число\ редукторов$, где % – операция деления по модулю, а $h(key)$ – хеш-значение ключа. Пара ключ-значение передается редуктору с номером $f(k)$. Пользователь может определить другие функции разбиения для поддержки более сложного поведения;
7) *модуль группировки* определяет, как объединять данные, полученные от разных процессов отображения в одну отсортированную последовательность на этапе редукции. Задание функции группировки, зависящей от выходного ключа преобразователя, позволяет осуществлять объединение данных более гибко. Например, если выходной ключ преобразователя состоит из нескольких атрибутов (sourceIP, destURL), то функция группировки может сравнивать только подмножество атрибутов, скажем (sourceIP). Тогда функция `reduce` в модуле редукции будет применяться к парам ключ-значение с одним и тем же sourceIP.

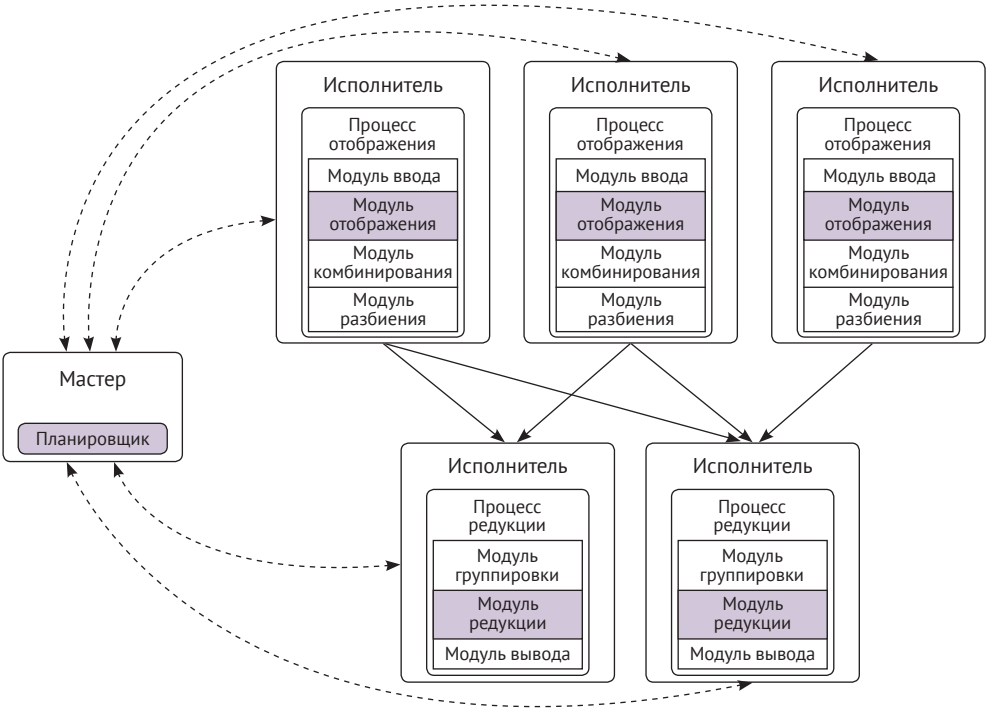


Рис. 10.5 ❖ Архитектура мастер-исполнитель каркаса MapReduce

Поскольку заявленная цель системы MapReduce – масштабирование на большое число обрабатывающих узлов, необходимо эффективно поддерживать отказоустойчивость. Если задача отображения или редукции завершается с ошибкой, то на другой машине создается другая задача для выполнения той же работы. Поскольку преобразователь хранит результаты локально, даже успешно завершенную задачу иногда приходится перезапускать в случае отказа узла. С другой стороны, так как редуктор сохраняет результаты в распределенном хранилище, успешно завершенную задачу редукции не придется перезапускать, если узел выйдет из строя.

10.2.1.2. Языки высокого уровня для MapReduce

В основе философии MapReduce лежит идея предложить гибкий каркас для решения различных задач. Поэтому MapReduce не включает языка запросов в расчете на то, что пользователи сами реализуют специализированные функции `map()` и `reduce()`. Гибкость при этом обеспечивается, но ценой сложности разработки приложений. Чтобы упростить использование MapReduce, было разработано несколько языков высокого уровня: декларативных (HiveQL, Tenzing, JAQL), потоковых (Pig Latin), процедурных (Sawzall). Существует также библиотека на Java (FlumeJava) и декларативные языки для машинного обучения (SystemML). Но с точки зрения баз данных, наибольший интерес представляют, пожалуй, декларативные языки. Несмотря на раз-

личия, все они обычно устроены похоже (рис. 10.6). На верхнем уровне находятся интерфейсы запросов, например: командная строка, веб-интерфейс или сервер JDBC/ODBC. В настоящее время только Hive поддерживает все эти интерфейсы. После того как через какой-то интерфейс пришел запрос, компилятор запросов разбирает его и генерирует логический план, используя метаданные. Затем к этому плану применяется оптимизация, основанная на правилах, например протолкнуть проекцию вниз. И наконец, план преобразуется в ориентированный ациклический граф (ОАГ), состоящий из заданий MapReduce, которые по очереди передаются движку выполнения.

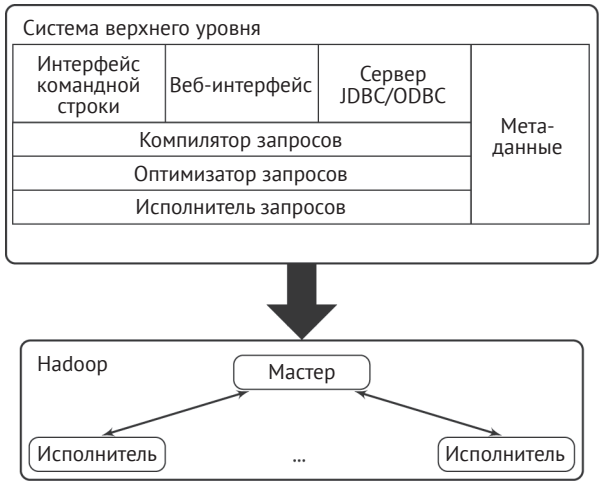


Рис. 10.6 ❖ Архитектура реализации декларативного языка запросов

10.2.1.3. Реализация операторов базы данных в MapReduce

Если реализацию MapReduce, в частности Hadoop, предполагается использовать для управления данными, выходящего за пределы «естественно распараллеливаемых» приложений, то важно реализовать типичные операторы базы данных, и этому предмету посвящен ряд научных работ. Такие простые операторы, как выборка и проекция, легко поддержать с помощью функции `map`, но другие – тета-соединение, эквисоединение, соединение нескольких отношений – требуют значительных усилий. В этом разделе мы рассмотрим подобные реализации.

Операции проецирования и выборки легко реализовать, добавив несколько условий в функцию `map`, чтобы отфильтровать ненужные столбцы и кортежи. Агрегирование легко реализуется с помощью функций `map()` и `reduce()`; на рис. 10.7 показан поток данных в задании MapReduce для оператора агрегирования. Преобразователь выделяет ключ агрегирования (Aid) из каждого входного кортежа (преобразованного в пару ключ-значение). Кортежи с одинаковыми ключами агрегирования попадают одному и тому же редуктору, который применяет к ним функцию агрегирования (например, sum или min).

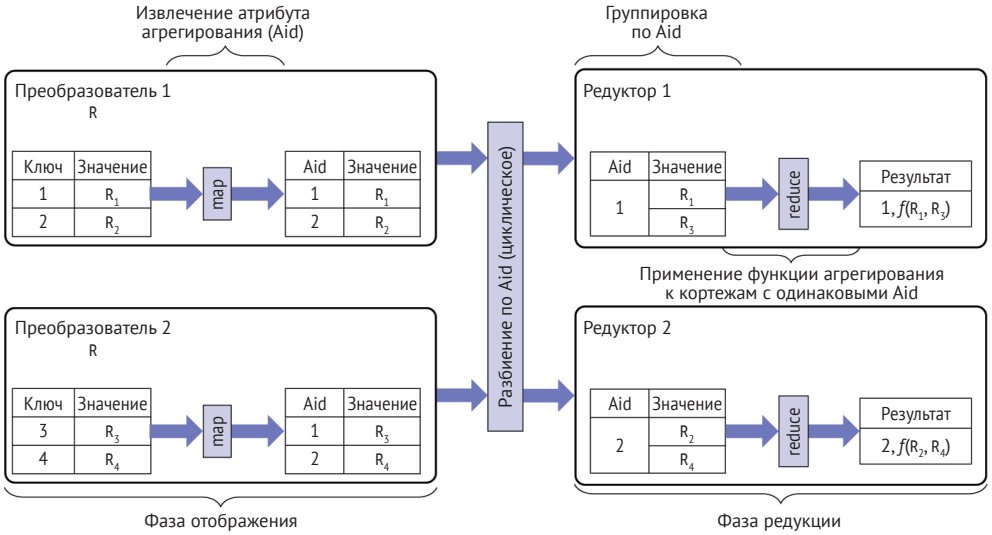


Рис. 10.7 ❖ Поток данных агрегирования

Наибольший интерес вызвала реализация оператора соединения, поскольку он один из самых дорогих, и удачная реализация может дать значительное повышение производительности. Сводка существующих алгоритмов соединения приведена на рис. 10.8. В качестве примеров мы опишем тета-соединение и эквисоединение.

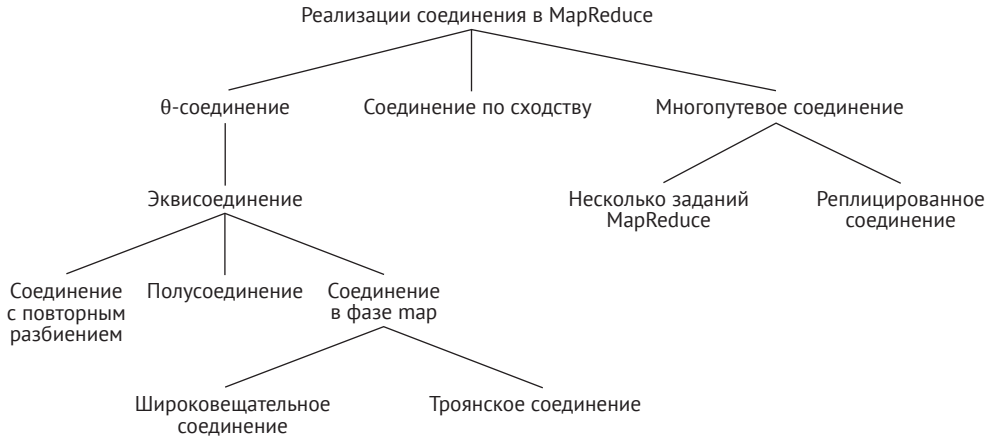


Рис. 10.8 ❖ Реализации соединения для каркаса MapReduce

Напомним, что тета-соединением (θ-соединением) называется оператор соединения, в котором условие θ – один из операторов $\{<, \leqslant, =, \geqslant, >, \neq\}$. Бинарное (естественное) соединение отношений `R(A, B)` и `S(B, C)` можно выполнить в MapReduce следующим образом. Отношение `R` секционируется, и каждой секции назначается множество преобразователей. Каждый преобразователь

принимает кортежи ⟨a, b⟩ и преобразует их в список пар ключ-значение вида (b, ⟨a, R⟩), где ключом является атрибут соединения, а значение включает имя отношения R. Эти пары тасуются и рассылаются редукторам, так чтобы все пары с одинаковыми ключами соединения попали одному редуктору. Такая же процедура применяется к отношению S. Затем каждый редуктор соединяет кортежи R с кортежами S (поскольку значение содержит имя отношения, то кортежи R будут соединяться только с кортежами S, но не друг с другом).

Для эффективной реализации тета-соединения в MapReduce |R| × |S| кортежей необходимо равномерно распределить между *r* редукторами, так чтобы каждый редуктор генерировал примерно одинаковое число результатов: (|R| × |S|)/*r*. Это делает алгоритм 1-Bucket-Theta, который равномерно разбивает матрицу соединения на блоки (рис. 10.9) и назначает каждый блок ровно одному редуктору, чтобы не повторять вычисления. Одновременно этот алгоритм гарантирует, что все редукторы получат равное количество блоков, и тем самым балансирует нагрузку. На рис. 10.9 таблицы R и S разбиты на 4 одинаковые части, в результате получилась матрица из 16 блоков, сгруппированных в 4 региона. Каждый регион назначается одному редуктору.

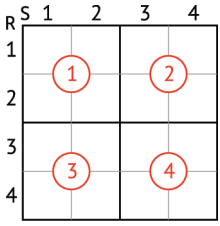


Рис. 10.9 ❖ Отображение матрицы декартова произведения на редукторы

На рис. 10.10 показан поток данных при тета-соединении, когда θ равно «≠», для случая, изображенного на рис. 10.9. Фазы отображения и редукции реализованы следующим образом:

1) *отображение*. Для каждого кортежа из R или S случайным образом выбирается номер строки или столбца (назовем его *Bid*) от 1 до количества регионов (в нашем примере – 4), который будет играть роль выходного ключа. К кортежу конкатенируется признак, указывающий, из какой таблицы он взят, и результат объявляется значением функции `map()`. *Bid* определяет, какой строке или столбцу матрицы (на рис. 10.9) принадлежит кортеж. Выходные кортежи функции `map()` тасуются и раздаются всем редукторам (каждый редуктор соответствует одному региону), которые пересекают строку или столбец;
2) *редукция*. Кортежи из одной и той же таблицы группируются на основе признаков. Затем к обоим разбиениям применяется вычисление тета-соединения. Результаты, удовлетворяющие условию соединения (R.*key* ≠ S.*key*), записываются в хранилище. Поскольку каждому блоку соответствует только один редуктор, лишних результатов не генерируется. На рис. 10.9 16 блоков образуют 4 региона; на рис. 10.10 имеется 4 редукто-

ра, каждый из которых отвечает за один регион. Поскольку редуктор 1 отвечает за регион 1, все кортежи R, для которых Bid = 1 или 2, и все кортежи S, для которых Bid = 1 или 2, передаются ему. Аналогично редуктор 2 получает кортежи R, для которых Bid = 1 или 2, и кортежи S, для которых Bid = 3 или 4. Каждый редуктор разбивает множество полученных кортежей на две части в зависимости от происхождения и соединяет эти части.

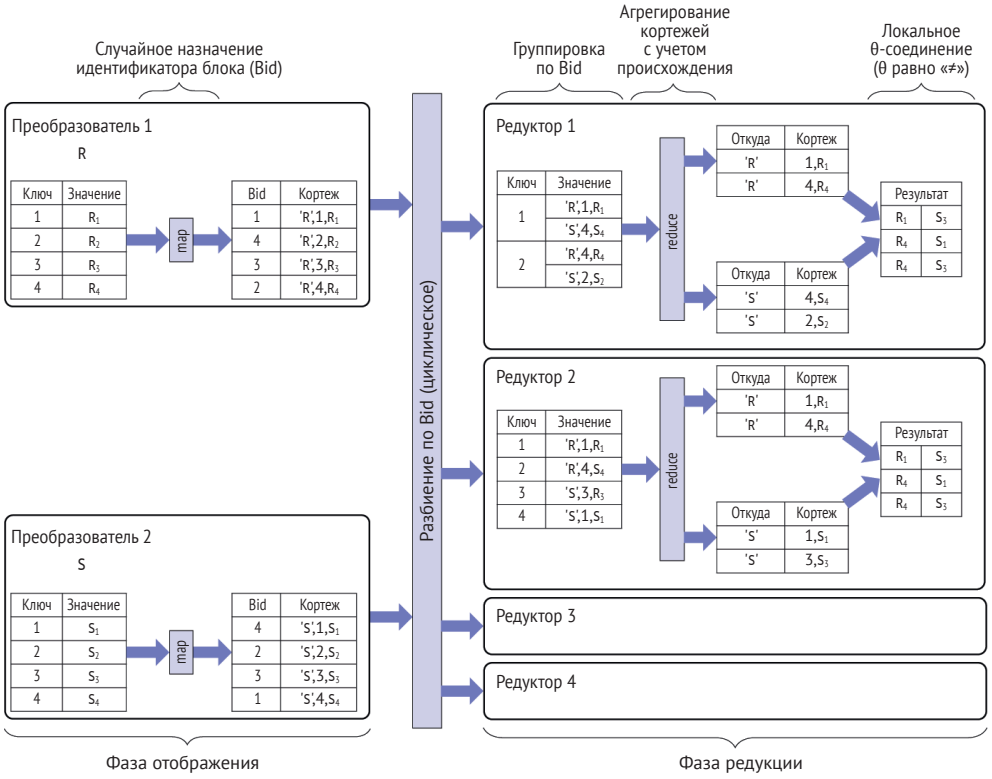


Рис. 10.10 ❖ Поток данных при θ-соединении (θ равно «≠»)

Рассмотрим теперь эквисоединение – частный случай θ-соединения при θ равно «=». Существует три способа реализации эквисоединения: с повторным разбиением, на базе полусоединения и только в фазе отображения. Соединение с повторным разбиением мы подробнее обсудим ниже. Соединение на базе полусоединения складывается из трех заданий MapReduce. Первое – полноценное задание, которое выделяет уникальные ключи соединения из одного отношения, скажем R; при этом задача отображения выделяет ключ соединения из каждого кортежа и передает одинаковые ключи одному и тому же редуктору, а задача редукции устраняет дубликаты ключей и сохраняет результаты в DFS в виде набора файлов (u_0, u_1, …, u_k). Во втором задании есть только фаза отображения, в которой порождаются результаты полусоединения $S' = S \ltimes R$. Поскольку файлы, в которых хранятся уникальные

ключи R, малы, они рассылаются каждому преобразователю и локально соединяются с частью S (*порцией данных*), назначенной этому преобразователю. Третье задание тоже состоит только из фазы отображения, в которой S′ рассылается всем преобразователям и локально соединяется с R.

Для соединения в фазе отображения редукции не требуется вовсе. Если внутреннее отношение гораздо меньше внешнего, то тасования можно избежать (как в широковещательном соединении), реализовав задачу отображения так, как третье задание в алгоритме на базе полусоединения. В предположении, что S – внутреннее, а R – внешнее отношение, каждый преобразователь загружает таблицу S целиком и строит в памяти хеш, а затем просматривает назначенную ему порцию данных R (т. е. R_i). Таким образом, выполняется локальное хеш-соединение S и R_i.

Соединение с повторным разбиением – алгоритм соединения, подразумеваемый в Hadoop по умолчанию. Две таблицы разбиваются в фазе отображения, после чего кортежи с одинаковыми ключами передаются одному редуктору, который их соединяет. Как показано на рис. 10.11, соединение с повторным разбиением можно реализовать как одно задание MapReduce.

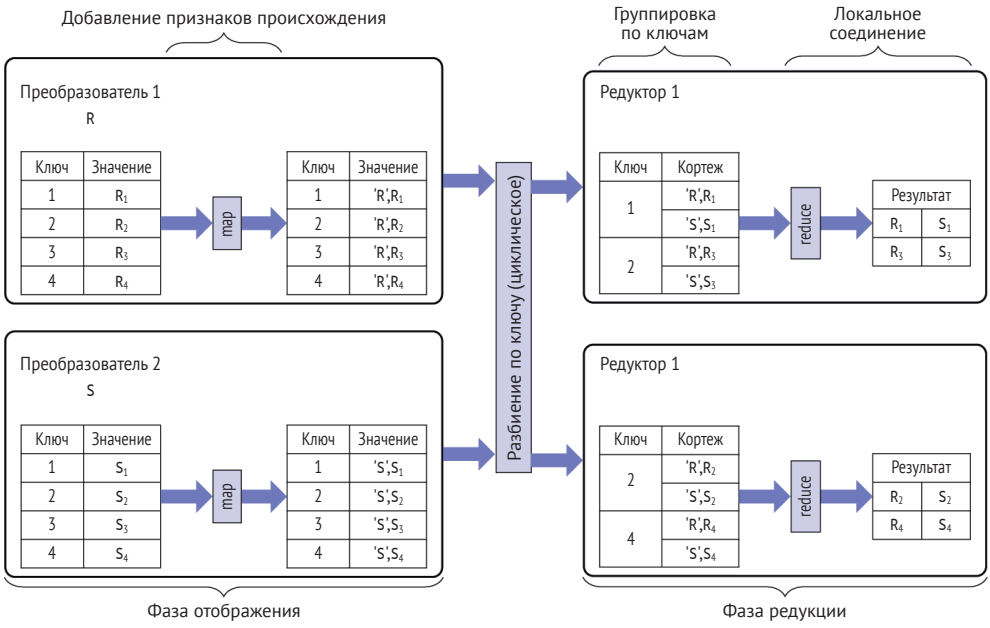


Рис. 10.11 ❖ Поток данных при соединении с повторным разбиением

1. *Отображение*. В фазе отображения создаются преобразователи двух типов, каждый из которых отвечает за обработку одной таблицы. Для каждого кортежа своей таблицы преобразователь выводит пару $(k, \langle t, v \rangle)$, где k – атрибут соединения, v – весь кортеж, а t – признак, показывающий, из какой таблицы этот кортеж взят. Точнее, фаза отображения состоит из следующих шагов:

a) последовательный просмотр данных в HDFS и генерирование пар ключ-значение;
b) сортировка результата отображения (т. е. множества пар ключ-значение). В фазе отображения выход каждого преобразователя необходимо отсортировать перед раздачей редукторам.

2. *Тасование*. После завершения задач отображения сгенерированные данные тасуются и передаются задачам редукции.
3. *Редукция*. Фаза редукции состоит из следующих шагов:
   a) объединение. Каждый редуктор объединяет полученные данные, применяя сортировку слиянием. Если памяти достаточно для обработки всех отсортированных последовательностей, то редуктору нужно только один раз прочитать данные и записать их в локальную файловую систему;
   b) соединение. После того как отсортированные последовательности объединены, редуктору нужно выполнить еще два действия, чтобы завершить соединение. Сначала множество кортежей с одинаковыми ключами разбивается на две части по признаку, показывающему, из какого отношения был взят кортеж. Затем обе части соединяются локально. В предположении, что количество кортежей с одним и тем же ключом невелико и все они помещаются в оперативную память, для этого нужно просмотреть отсортированную последовательность только один раз;
   c) запись в HDFS. Наконец, результаты, сгенерированные редуктором, записываются обратно в HDFS.

10.2.2. Обработка данных с помощью Spark

Базовый каркас MapReduce, рассмотренный в предыдущем разделе, плохо подходит для информационно емких приложений, в которых имеются итеративные вычисления, требующие цепочки из нескольких заданий MapReduce (например, добыча данных), или оперативное агрегирование. В этом разделе мы обсудим важное расширение MapReduce на такой класс приложений – систему Spark. Начнем с того, как можно было бы выполнить итеративное вычисление в базовой системе MapReduce и почему это вызывает проблемы.

На рис. 10.12 показано итеративное задание, обладающее двумя особенностями: (1) источник данных на каждой итерации состоит из переменной и постоянной частей – переменная часть включает файлы, сгенерированные предыдущими заданиями MapReduce (серые стрелки на рис. 10.12), а постоянная часть – это исходные входные файлы (черные стрелки); (2) в конце каждой итерации нужно проверить, достигнута ли точка остановки. В разных приложениях условия остановки различаются; в алгоритме кластеризации методом k средних, который мы обсудим в примере 10.2, проверяется, что внутрикластерная сумма квадратов расстояний достигла минимума, а в алгоритме PageRank (пример 10.4) – что вычисление рангов всех вершин сошлось. На этом рисунке видны три проблемы, с которыми сталкивается MapReduce

в задачах такого типа. Первая заключается в том, что после каждого задания (итерации) промежуточные результаты нужно записать в распределенную файловую систему (например, HDFS), а в начале следующего задания (итерации) снова прочитать. Вторая проблема в том, что нет никаких гарантий, что последующие задания будут назначены тем же машинам. Поэтому постоянные данные, которые не зависят от итерации, невозможно оставить в узлах-исполнителях, а приходится читать заново. И третья проблема в том, что в конце каждой итерации необходимо дополнительное задание, которое сравнивает результаты текущего и предыдущего заданий (проверка сходимости). Из-за этих – довольно высоких – накладных расходов использование MapReduce для таких приложений неэффективно. Для решения проблемы было предложено несколько подходов, одни из которых применимы к конкретной задаче, например анализ графов, который мы обсудим в следующем разделе, а другие, в частности Spark, носят более общий характер.

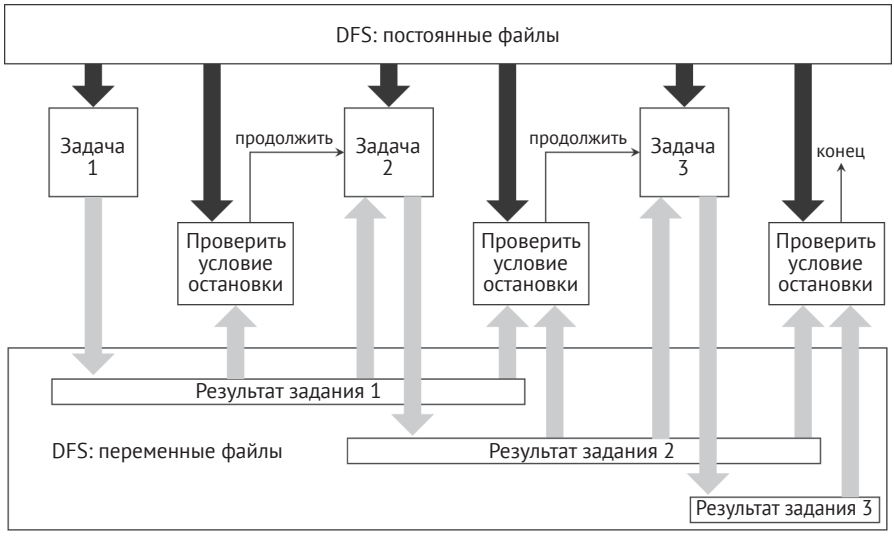


Рис. 10.12 ❖ Итеративные вычисления в MapReduce

Пример рабочей нагрузки, вызывающей проблемы в MapReduce, – алгоритм кластеризации k средних, который часто применяется для анализа больших данных. В примере 10.2 мы опишем этот алгоритм и обсудим, с какими трудностями сталкивается попытка реализовать его с помощью MapReduce.

Пример 10.2. Алгоритм k средних принимает множество значений X и разбивает его на k кластеров, помещая каждое $x_i \in X$ в кластер C_j, центроид которого отстоит от x_i на наименьшее расстояние. Центроидом кластера называется среднее всех принадлежащих ему элементов. Под расстоянием понимается внутрикластерная сумма квадратов, т. е. величина $\sum_{j=1}^{k}\sum_{x_i \in C_j}(x_i - \mu_j)^2$, где μ_j – центроид кластера C_j. Таким образом, мы пытаемся найти такое распределение по кластерам, которое доставляет минимум этой функции для любого x_i.

Стандартный алгоритм k средних принимает множество значений $X = \{x_1, x_2, \ldots, x_r\}$ и начальное множество центроидов $M = \{\mu_1, \mu_2, \ldots, \mu_m\}$ (обычно $r \gg m$) и итеративно выполняет следующие три шага:

1) вычислить расстояние между каждым $x_i \in X$ и каждым центроидом $\mu_j \in M$ и отнести x_i к тому кластеру C_z, для которого вышеуказанная функция достигает минимума;
2) вычислить новое множество центроидов M в соответствии с новым распределением значений по кластерам;
3) для каждого кластера проверить, совпадают ли старое и новое значения центроида. Если да, алгоритм сошелся и останавливается. В противном случае перейти к шагу 1 и выполнить еще одну итерацию.

В MapReduce этот алгоритм реализуется прямолинейно: первый шаг выполняется в фазе отображения, когда каждый исполнитель (преобразователь) производит вычисления над подмножеством X, а второй шаг – в фазе редукции. Для третьего шага – проверки сходимости – нужно еще одно задание, как было сказано выше. Отметим, что всем преобразователям нужно полное множество центроидов M, поэтому на шаге 3 (проверка сходимости) необходимо разослать всем узлам новые центроиды, если сходимость еще не достигнута.

На этом примере отчетливо видны проблемы реализации итеративных заданий в MapReduce: результаты вычислений в конце каждой итерации (т. е. пересчитанные центроиды M и текущая конфигурация кластеров C) должны быть записаны в HDFS, откуда их смогут прочитать преобразователи и редукторы на следующей итерации; поскольку задание для следующей итерации может быть назначено любой машине, постоянные данные (т. е. X) необходимо повторно разбить и заново прочитать. И еще в конце каждой итерации имеется дополнительное задание для проверки сходимости. ♦

В Spark эта проблема MapReduce решается путем введения абстракции разделения данных между несколькими стадиями итеративного вычисления. Эта абстракция называется *гибкий распределенный набор данных* (resilient distributed dataset – RDD). Эффективное разделение достигается благодаря двум свойствам RDD. Во-первых, гарантируется, что секции, назначенные каждому узлу-исполнителю, не меняются от итерации к итерации, поэтому можно обойтись без тасования данных; во-вторых, от записи и чтения из HDFS между итерациями можно отказаться, сохранив наборы RDD в памяти, – поскольку назначение работ исполнителям сохраняется при переходе к следующей итерации, это возможно.

RDD создается пользователем, который решает, как разбить его между узлами кластера и где хранить – на диске или в оперативной памяти. Если RDD хранится в памяти, то он играет роль кеша рабочего набора в приложении. RDD – неизменяемая (допускающая только чтение) коллекция записей; для обновления RDD необходимо выполнить *преобразование* (например, `map()`, `filter()`, `groupByKey()`), в результате чего создается новый RDD. Таким образом, RDD можно создать либо путем чтения из файловой системы, либо из другого RDD с помощью преобразования.

Пример 10.3. Рассмотрим, как реализовать кластеризацию методом k средних (пример 10.2) в Spark. Мы не станем описывать алгоритм целиком[1], а лишь расскажем, как решаются вышеупомянутые проблемы.

1. Создать RDD для постоянных данных (множество X) и кешировать его в памяти, чтобы не выполнять ввод-вывод между итерациями.
2. Создать RDD для переменных данных (множество центроидов M).
3. Вычислить расстояния между каждым $x_i \in X$ и каждым $\mu_j \in M$; сохранить эти расстояния в виде RDD D.
4. На основе D создать новый RDD, который включает все x_i и для каждого из них центроид μ_j, отстоящий на минимальное расстояние.
5. Создать RDD M_{new}, который содержит среднее значений x_i, отнесенных к каждому μ_j.
6. Сравнить M и M_{new} на предмет установления сходимости.
7. Если алгоритм еще не сошелся, положить $M \leftarrow M_{new}$. Перегружать постоянные данные не нужно, можно переходить сразу к шагу 3. ♦

Важный аспект RDD – нужно ли сохранять его между итерациями (или заданиями MapReduce). Если это желательно, то к RDD можно применить одно из двух преобразований: `cache` или `persist`. Если RDD должен оставаться в оперативной памяти, то используется `cache`; если же нужно гибко указать «уровень сохранения» (только диск, диск и оперативная память и т. д.), то используется `persist` с параметрами (по умолчанию подразумевается сохранение в памяти).

RDD вычисляется лениво – когда программа затребует *действие*. Действия (например, `collect()`, `count()`) отличаются от преобразований тем, что RDD материализуется в момент выполнения первого действия, и это действие выполняется во всех узлах, где размещены части RDD. Этот аспект мы обсудим ниже.

Теперь рассмотрим поток выполнения Spark-программы, изображенный на рис. 10.13. Первым делом программа создает RDD из данных, хранящихся в HDFS. Затем в зависимости от решения пользователя – кешировать или сохранять RDD – система совершает подготовительные действия. Далее могут быть дополнительные преобразования для создания других RDD, и для каждого из них принимается решение о кешировании или сохранении. Наконец, начинается обработка действий, указанных в программе. Как уже было сказано, первое же действие приводит к материализации RDD, после чего применяется к нему. В процессе обработки действия и задания повторяются.

Обсудим, как Spark поддерживает выполнение программ, написанных в соответствии с концепцией RDD. Spark ожидает, что имеется управляющая машина, на которой работает программа-драйвер. Драйвер генерирует запрошенные пользователем RDD и в момент выполнения первого действия материализует RDD, распределяет его между исполнителями и выполняет действия на каждом исполнителе. Управляющая машина соответствует узлу-мастеру в MapReduce, а драйвер – функции планирования. Следуя решению

---

[1] С полной реализацией того варианта алгоритма, который обсуждался выше, можно познакомиться по адресу https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/clustering/KMeans.scala.

о кешировании/сохранении для каждого RDD, драйвер просит исполнителей выполнить соответствующие действия. Когда исполнители сообщают о завершении действия, драйвер инициирует следующее действие. Применяются обычные оптимизации в части управления RDD, обращения с отстающими узлами и т. д., но все это выходит за рамки книги.

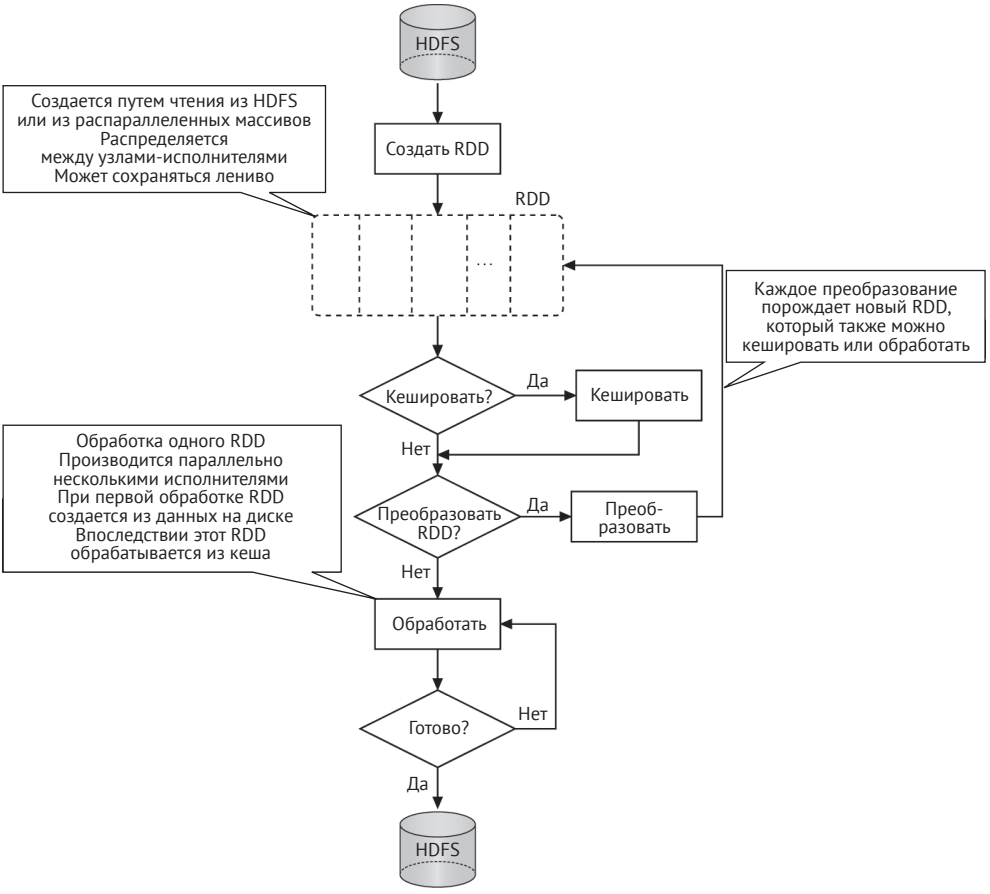


Рис. 10.13 ❖ Поток выполнения Spark-программы

Spark добавляет к стандартному MapReduce отказоустойчивость, запоминая родословную RDD. Иначе говоря, он хранит граф, в котором указано, как каждый RDD генерировался на основе других. Родословная конструируется как объект и сохраняется на диске на случай восстановления. Если произойдет сбой и RDD будет потерян, то его можно будет заново вычислить по родословной. Кроме того, как уже было сказано, каждый RDD распределяется между исполнителями, поэтому утрата, скорее всего, ограничится несколькими частями, и только их и придется пересчитывать.

Важная цель Spark – не просто реализовать эталонную архитектуру, которую мы обсуждали с общей точки зрения, но и предоставить целую эко-

систему. Поэтому поверх Spark была построена реляционная СУБД (Spark SQL), система потоковой обработки данных (Spark Streaming) и система для обработки графов (GraphX). В следующих разделах мы обсудим Spark Stream и GraphX.

10.3. Управление потоковыми данными

Традиционные системы управления данными, которые мы рассматривали до сих пор, имеют дело со множеством неупорядоченных и относительно статических объектов, т. е. операции вставки, обновления и удаления производятся реже, чем запросы. Иногда можно встретить термин *моментальные базы данных* (snapshot database), поскольку хранится снимок значений объектов данных в данный момент времени[1]. Запросы к таким системам задаются явно, а ответ отражает текущее состояние базы данных. То есть парадигму можно описать как выполнение *скоротечных* запросов к *долговременным* данным.

Но появился класс приложений, которые не укладываются в такую модель данных и парадигму запросов. Среди прочих, к ним относятся сети датчиков, анализ трафика в сетях, интернет вещей (IoT), биржевые ленты, онлайновая торговля и аукционы, приложения для анализа журналов (в частности, запросов к веб-сайтам и записей телефонных звонков). В таких приложениях данные генерируются в режиме реального времени и выглядят как бесконечная последовательность (поток) значений. Они называются приложениями *потоковой обработки данных*, или *потоковыми приложениями*. В этом разделе мы обсудим системы, поддерживающие такие приложения. Потоковые приложения отражают высокий темп порождения больших данных.

Существует два типа систем для обработки потоков данных: *системы управления потоками данных* (СУПД, англ. DSMS) предоставляют функциональность типичной СУБД, включая и язык запросов (декларативный или основанный на потоках данных), а *системы обработки потоков данных* (СОПД, англ. DSPS) не претендуют на полноценную функциональность СУБД. Первые системы относились к категории СУПД, некоторые включали декларативные языки (STREAM, Gigascope, TelegraphCQ), тогда как другие (например, Aurora и ее распределенная версия Borealis) – языки потоков данных. Более современные системы относятся к категории СОПД (Apache Storm, Heron, Spark Streaming, Flink, MillWheel, TimeStream). Многие ранние СУПД размещались на одной машине (за исключением Borealis), но все современные СОПД являются распределенными или параллельными.

Фундаментальное предположение модели потока данных заключается в том, что новые данные генерируются непрерывно и в фиксированном порядке, хотя темп поступления данных может различаться: от миллионов событий в секунду (мониторинг интернет-трафика) до нескольких событий

[1] Напомним, что в хранилищах данных обычно хранятся исторические данные для анализа тенденций. Большинство рассмотренных нами систем относятся к категории OLTP и имеют дело с моментальными снимками.

в час (считываний показаний датчиков температуры и влажности на метеорологической станции). Порядок потока данных может быть неявным (время поступления на место обработки) или явным (время генерации, указанное во *временной метке*, которую источник прикрепляет к каждому элементу данных). Ввиду этих предположений к системам потоков данных (СПД, англ. DSS)[1] предъявляются следующие требования:

1) вычисления, осуществляемые СПД, по большей части инициированы проталкиванием, т. е. управляются данными. Новые данные непрерывно (или периодически) поступают в систему для обработки. С другой стороны, в традиционных СУБД модель вычислений в основном основана на вытягивании, т. е. управляется запросами, в ответ на которые запускается обработка;
2) как следствие запросы и рабочая нагрузка в СПД обычно *постоянные* (употребляются также термины *непрерывные* и *долгосрочные*), т. е. задаются один раз, но остаются активными в течение длительного периода времени. Это означает, что обновленные результаты порождаются непрерывно – потоком. Конечно, такие системы могут принимать и выполнять однократные вопросы, как в традиционных СУБД, но их отличительной особенностью является наличие постоянных запросов;
3) предполагается, что поток данных бесконечен или, по крайней мере, его длина неизвестна. Поэтому нет возможности прибегнуть к обычной схеме и сохранить все данные, перед тем как приступить к выполнению запросов; запросы нужно обрабатывать по мере поступления данных. В некоторых системах применяется *модель непрерывной обработки*, когда каждый новый элемент данных обрабатывается сразу после поступления (например, Apache Storm, Heron). В других используется *модель оконной обработки*, когда входящие данные собираются в пакеты, а затем весь пакет обрабатывается (например, STREAM, Spark Streaming). С точки зрения пользователя, недавно поступившие данные могут быть более интересны и полезны, поэтому окно определяется на уровне приложения. Системы с моделью непрерывной обработки могут предоставлять и обычно предоставляют средства для обработки окон. Так что, с точки зрения пользователя, это «два в одном». Кроме того, система может реализовывать окна на внутреннем уровне, чтобы избежать блокирующих операций, – и скоро мы это увидим;
4) условия в системе на протяжении *времени жизни* постоянного запроса могут быть нестабильны. Например, темп поступления данных может колебаться, количество запросов тоже меняется.

Абстрактная эталонная архитектура СПД с одним узлом показана на рис. 10.14. Данные поступают из одного или нескольких внешних источников. Входной монитор регулирует темп ввода и может отбрасывать часть элементов, если система не успевает обрабатывать. Данные обычно хранятся в трех разделах: временное рабочее хранилище (например, для оконных запросов, которые мы обсудим ниже), сводное хранилище для сводной информации о состоянии потока (необязательно, поскольку некоторые системы не

[1] Мы будем использовать этот более общий термин, когда разделение на СУПД и СОПД несущественно.

раскрывают состояние потока приложениям, поэтому не нуждаются в его хранении) и статическое хранилище метаданных (например, о физическом местоположении каждого источника). Постоянные запросы регистрируются в репозитории запросов и объединяются в группы для совместной обработки, что не мешает предъявлять однократные запросы к текущему состоянию системы. Процессор запросов взаимодействует с входным монитором и может повторно оптимизировать планы выполнения запросов в ответ на изменяющийся темп поступления данных. Результаты потоком отправляются пользователям или сохраняются во временном буфере. Пользователи могут уточнять свои запросы, исходя из недавних результатов. В распределенной или параллельной СПД эта архитектура может быть реплицирована в каждом узле и содержать дополнительные компоненты для коммуникации и управления распределенными данными.

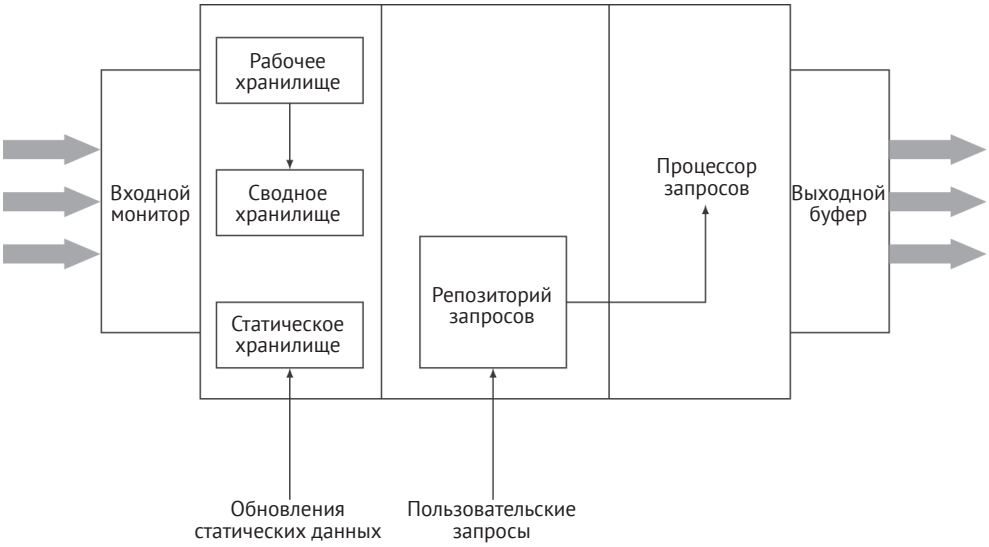


Рис. 10.14 ❖ Абстрактная эталонная архитектура системы управления потоками данных

10.3.1. Потоковые модели, языки и операторы

Теперь займемся фундаментальными вопросами, возникающими при рассмотрении моделей потоковых систем. На эту тему существует обширная литература, и в разделе «Библиографические замечания» приведены некоторые ссылки. Но сейчас наша цель – объяснить важнейшие концепции, без которых невозможно понять последующее обсуждение.

10.3.1.1. Модели данных

Поток данных – это последовательность снабженных временными метками элементов, поступающих в определенном порядке, которая допускает толь-

ко дописывание в конец. Это общепринятое определение, но существуют и ослабленные варианты; например, можно рассмотреть *исправленные кортежи*, которые заменяют ранее переданные (вероятно, ошибочные) данные, тогда последовательность уже нельзя назвать допускающей только дописывание. В системах публикации-подписки, когда данные порождаются несколькими источниками, а потребляются теми, кто на них подписался, поток данных можно представлять себе как последовательность непрерывно возникающих событий. Поскольку данные могут поступать пачками, поток можно моделировать как последовательность множеств (или мешков) элементов, в которой каждое множество содержит элементы, поступившие в течение некоторого промежутка времени (не предполагается никакого порядка между элементами, поступившими точно в одно и то же время). В реляционных потоковых моделях (например, STREAM) отдельные элементы имеют вид реляционных кортежей, поэтому все кортежи, поступающие в одном потоке, описываются одной и той же схемой. В объектных моделях (например, COUGAR и Tribeca) источники и типы элементов могут быть экземплярами (иерархических) типов данных с ассоциированными методами. В более современных системах, например Apache Storm, Spark Streaming и других, данные могут иметь произвольную форму, определяемую приложением, поэтому иногда употребляется термин *полезная нагрузка*. К элементу потока может быть прикреплена явная присвоенная источником временная метка и неявная метка, назначаемая СУПД в момент поступления, так что каждый элемент имеет вид ⟨временная метка, полезная нагрузка⟩. В любом случае атрибут временной метки может быть или не быть частью схемы потока, а потому может быть виден или не виден пользователям. Элементы потока могут прибывать не по порядку (если используются явные временные метки) и (или) в предварительно обработанном виде. Например, вместо того чтобы распространять заголовок каждого IP-пакета, можно отправлять одно значение (или несколько частично агрегированных значений), описывающее продолжительность соединения между двумя IP-адресами и количество переданных байтов.

Было определено несколько классификаций оконных моделей, но наиболее важны и распространены два критерия:

1) *направление движения концевых точек.* Две фиксированные концевые точки определяют *фиксированное окно*, две скользящие концевые точки (вперед или назад – с заменой старых элементов новыми по мере их поступления) – *скользящее окно*, а одна фиксированная и одна скользящая (вперед или назад) – *расширяющееся окно*;
2) *определение размера окна.* Логические, или *временные*, окна определяются в терминах интервала времени, а физические (*считающие*) – в терминах количества элементов данных. Кроме того, можно определить *расщепленное окно*, если разбить окно на группы и определить отдельное считающее окно для каждой группы. Самым общим является *предикатное окно*, содержимое которого определяется произвольным предикатом, например все пакеты из всех открытых в данный момент TCP-соединений. Предикатное окно аналогично материализованному представлению и называется также *сеансовым*, или *определенным пользователем*, *окном*.

В этой классификации самыми важными являются временные и считающие скользящие окна. Именно к ним было приковано наибольшее внимание, их мы и будем рассматривать ниже.

10.3.1.2. Модели и языки потоковых запросов

Важный вопрос – какова семантика постоянных запросов, т. е. как они генерируют ответы. Постоянные запросы могут быть монотонными и немонотонными. *Монотонным* называется запрос, результаты которого можно обновлять инкрементно, т. е. достаточно пересчитывать запрос для вновь поступающих элементов и добавлять удовлетворяющие условиям кортежи к результату. Поэтому ответом на монотонный постоянный запрос является непрерывный поток результатов, допускающий только дописывание в конец. Иногда выход периодически обновляется путем дописывания пачки новых результатов. *Немонотонные* запросы могут порождать результаты, которые перестают быть верными по мере добавления новых данных и изменения или удаления существующих. Следовательно, их необходимо пересчитывать с нуля при каждом ответе.

Как уже отмечалось, СУПД предоставляют язык запросов. Существует две основные парадигмы запросов: декларативная и процедурная. Декларативные языки обладают похожим на SQL синтаксисом, но потоковой семантикой. В этот класс попадают языки CQL, GSQL и StreaQuel. В *процедурных языках* запросы конструируются путем построения ациклического графа операторов (как в Aurora).

Языки, поддерживающие оконное выполнение, предоставляют два примитива: `size` и `slide`. Первый задает ширину окна, второй – частоту сдвига окна. Например, для временного скользящего окна `size=10min`, `slide=5sec` означает, что нас интересуют данные за последние 10 минут, а окно сдвигается каждые 5 секунд. Эти параметры влияют на способ управления содержимым окна, мы обсудим их в разделе 10.3.2.1.

10.3.1.3. Потоковые операторы и их реализация

Приложения, генерирующие потоки данных, похожи также типами выполняемых операций. Перечислим основные операции, применяемые к потоковым данным.

- **Выборка.** Все потоковые приложения поддерживают сложную фильтрацию.
- **Сложное агрегирование.** Для выявления тенденций в данных необходимо сложное агрегирование (например, сравнение минимума со скользящим средним), частые запросы данных и т. д.
- **Мультиплексирование и демультиплексирование.** Иногда физический поток необходимо разложить на несколько логических потоков, и, наоборот, логические потоки необходимо объединить в один физический (аналоги операторов `group-by` и `union` соответственно).
- **Анализ потока.** Такие операции, как сопоставление с образцом, поиск похожих и прогнозирование, необходимы для онлайнового анализа потоковых данных.

- **Соединения.** Необходимо поддержать соединение нескольких потоков и соединение потоков со статическими метаданными.
- **Оконные запросы.** Все вышеперечисленные запросы можно ограничить данными внутри окна (например, за последние 24 часа или из последних ста пакетов).

На первый взгляд, это напоминает обыкновенные реляционные операторы запросов, но их реализация и оптимизация ставят новые проблемы, которые мы обсудим ниже.

Некоторые операторы не запоминают состояния (например, проекция и выборка), поэтому можно воспользоваться их реляционными реализациями без существенных модификаций. На рис. 10.15a в качестве примера показана реализация оператора выборки. Входные кортежи просто фильтруются на основе условия выборки.

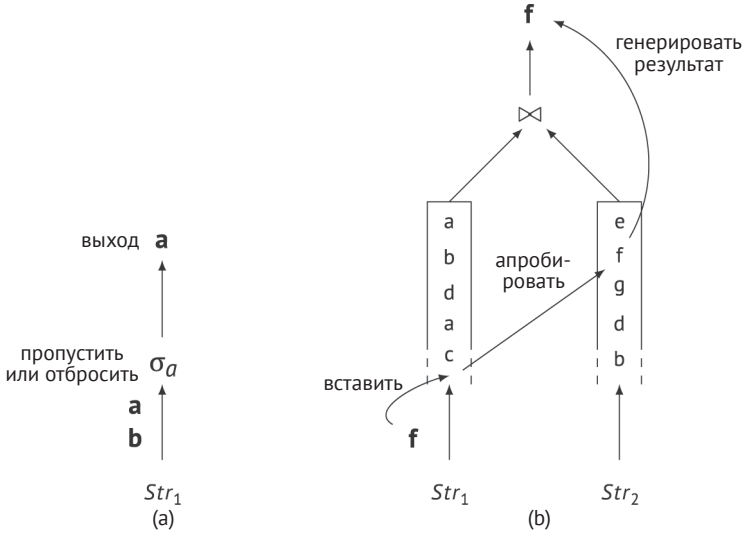


Рис. 10.15 ❖ Операторы непрерывных запросов: (a) выборка; (b) соединение

Однако операторы, запоминающие состояние (например, операторы соединения), в реляционных реализациях характеризуются блокирующим поведением, что в СПД неприемлемо. Например, прежде чем перейти к следующему кортежу, алгоритму вложенных циклов, возможно, потребуется просмотреть все внутреннее отношение и сравнить каждый его кортеж с текущим внешним кортежем. Но поток данных бесконечен, поэтому такая блокировка вызывает проблемы. Доказано, что запрос является монотонным тогда и только тогда, когда он *неблокирующий*, т. е. не должен ждать признака конца ввода, перед тем как приступить к порождению результатов. У некоторых операторов имеются неблокирующие аналоги, в частности у соединения и простых агрегатов. Например, неблокирующий алгоритм конвейерного симметричного хеш-соединения (двух символьных потоков Str_1 и Str_2) ди-

намически строит хеш-таблицы для Str_1 и Str_2 (см. рис. 10.15b). Хеш-таблицы хранятся в оперативной памяти. Когда поступает кортеж из какого-то отношения, он вставляется в свою таблицу, а другая таблица апробируется на наличие соответствия, и если таковое найдено, то порождаются результаты, содержащие новый кортеж. Соединение более двух потоков и соединение потока со статическим отношением – простое обобщение описанной процедуры. В первом случае при поступлении кортежа из одного источника все остальные источники апробируются в определенном порядке. Во втором – при поступлении нового кортежа из потока апробируется отношение. Поскольку строить хеш-таблицы для бесконечных потоков невозможно, в большинстве СУПД поддерживаются только оконные соединения, когда для каждого входного потока определено окно и соединение вычисляется для данных из этих окон в соответствии с семантикой окна.

Чтобы избавиться от блокировки при вычислении оператора запроса, можно реализовать его в инкрементной форме, ограничив окном и воспользовавшись потоковыми ограничениями, например *прерывателями* (punctuation) – ограничениями (представленными в виде элементов данных), задающими условия для всех будущих элементов. Чуть ниже мы еще вернемся к прерывателям. Операторы скользящих окон обрабатывают события двух типов: поступление новых данных и истечение срока хранения старых данных. Мы подробно обсудим их в следующем разделе при рассмотрении проблем обработки запросов.

10.3.2. Обработка запросов к потокам данных

Методология обработки запросов к потоковым данным похожа на реляционный случай, пусть и с некоторыми модификациями: декларативные запросы транслируются в планы выполнения, которые отображают заданные в запросе логические операторы в физические реализации. Однако при рассмотрении деталей обнаруживается ряд различий.

Важное отличие – присутствие постоянных запросов и тот факт, что операции потребляют данные, которые проталкиваются в план источниками, а не вытягивают их из источников, как в традиционных СУБД. Кроме того, как уже было сказано, операции могут быть (и часто бывают) более сложными, чем реляционные операторы, и включают определенные пользователем функции (UDF). Запрос позволяет источнику включать в план запроса данные и операции по их получению. При простой стратегии планирования каждой операции назначается квант времени, в течение которого она извлекает кортежи из своей входной очереди (или нескольких очередей), обрабатывает их в порядке временных меток и помещает выходные кортежи во входную очередь следующей операции (рис. 10.16).

Выше мы говорили, что СПД могут следовать модели непрерывной обработки или оконного выполнения. Во втором случае важно, как организовано управление окнами, точнее как элементы данных добавляются в текущее окно и удаляются из него. Это еще одно отличие от реляционных СУБД, которое мы обсудим в разделе 10.3.2.1. Две другие особенности потоковых систем –

управление нагрузкой в случае, когда темп поступления данных настолько велик, что система не справляется, и обращение с данными, поступающими не по порядку, – обсуждаются соответственно в разделах 10.3.2.2 и 10.3.2.3. Наконец, постоянные запросы открывают дополнительные возможности для многозапросной обработки, эту тему мы обсудим в разделе 10.3.2.4.

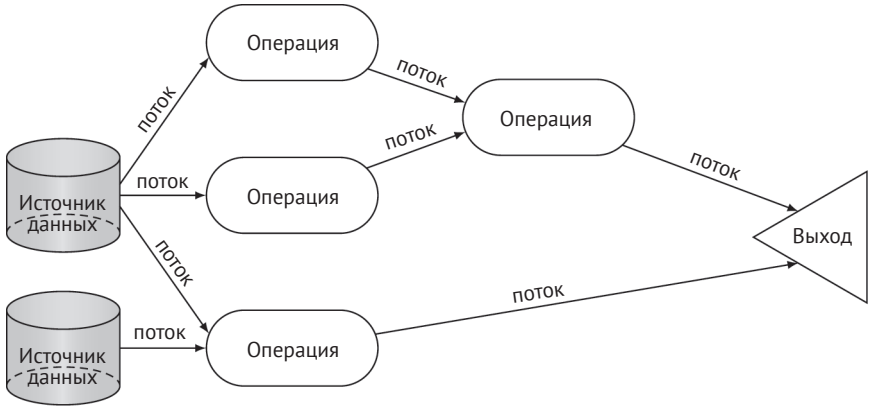


Рис. 10.16 ❖ Пример плана потокового запроса

Пути распределенных и параллельных СПД расходятся, как и в реляционном случае. В распределенной системе главное – секционировать план выполнения по нескольким обрабатывающим узлам в зависимости от данных, находящихся в узле. Секционирование плана запроса заключается в назначении операторов запроса узлам и со временем может потребовать перебалансировки. Возникающие при этом вопросы аналогичны тем, что возникают в распределенных СУБД; мы подробно обсуждали их выше. В параллельных системах обычно применяется распараллеливание по данным, когда поток разбивается на части и каждый узел выполняет один и тот же запрос над подмножеством данных. Большинство современных систем отдают предпочтение второму подходу, который мы обсудим в разделе 10.3.2.5.

10.3.2.1. Выполнение оконного запроса

Выше мы отмечали, что при оконном выполнении система должна решить вопрос о поступлении новых данных и истечении срока хранения старых. Действия в том и в другом случае зависят от оператора. Новые данные могут порождать новые результаты (например, в случае соединения) или приводить к удалению ранее сгенерированных результатов (как в случае отрицания). Кроме того, истечение срока хранения данных может потребовать удаления данных из результата (в случае агрегирования) или пополнения результата (в случае устранения дубликатов или отрицания). Заметим, что мы здесь не обсуждаем случай, когда элемент данных удаляется приложением. Речь идет только об удалении элементов из результатов запроса вследствие оконных операций.

Рассмотрим, к примеру, соединение скользящих окон: новые данные, поступающие из одного входа, апробируют состояние другого входа, как при соединении бесконечных потоков. Дополнительно данные с истекшим сроком хранения удаляются из состояния.

Истечение срока хранения данных во временном окне определяется просто: срок элемента данных истек, если его временная метка вышла за пределы диапазона окна.

В считающем окне количество элементов данных всегда постоянно. Поэтому для реализации истечения срока хранения можно перезаписывать самые старые элементы новыми. Однако если оператор хранит состояние, соответствующее результату операции соединения считающих окон, то размер состояния может изменяться в зависимости от значений атрибутов соединения в новых кортежах.

Вообще говоря, существует два метода обработки запросов к скользящим окнам и обслуживания состояния: метод вычитающих (negative) кортежей и прямой метод. В первом случае каждому окну, упоминаемому в запросе, сопоставляется оператор, который явно генерирует вычитающий кортеж для каждого события истечения строка хранения, помимо проталкивания вновь поступающих кортежей в план запроса. Таким образом, каждое окно должно быть материализовано, чтобы можно было создать подходящие вычитающие кортежи. Вычитающие кортежи пропускаются через план запроса и обрабатываются операторами так же, как регулярные, но при этом операторы должны еще удалять соответствующие «реальные» кортежи из своего состояния. Подход на основе вычитающих кортежей можно эффективно реализовать, используя хеш-таблицы для хранения состояния оператора; тогда кортеж с истекшим сроком хранения можно будет быстро найти при появлении вычитающего кортежа. Недостаток в том, что приходится обрабатывать вдвое больше кортежей, поскольку срок хранения каждого кортежа рано или поздно истекает, в результате чего генерируется вычитающий кортеж. Кроме того, в плане должны присутствовать дополнительные операторы, генерирующие вычитающие кортежи по мере сдвига окна.

Прямой метод позволяет обрабатывать запросы к временным окнам, не содержащие отрицания. Такие запросы обладают тем свойством, что время истечения срока хранения базовых кортежей и промежуточных результатов можно определить по временным меткам истечения, которые равны времени поступления плюс ширина окна. Поэтому операторы могут напрямую обращаться к своему состоянию и находить истекшие кортежи, не привлекая для этого вычитающих кортежей. У прямого метода нет накладных расходов на обработку вычитающих кортежей и не требуется хранить базовые окна, упоминаемые в запросе. Однако он может работать медленнее для запросов к нескольким окнам, потому что при вставке или удалении иногда необходим последовательный просмотр буферов состояния.

10.3.2.2. Управление нагрузкой

Темп поступления данных из потока может быть настолько велик, что все кортежи обрабатывать не удается, какие бы методы оптимизации (стати-

ческой или динамической) ни использовались. В таком случае применяется один из двух способов ограничения нагрузки: случайный или семантический. Во втором методе используются свойства потока или параметры качества обслуживания, чтобы отбрасывать кортежи, которые кажутся менее важными. В качестве примера семантического ограничения нагрузки рассмотрим приближенное соединение скользящих окон с целью получить результат максимального размера. Идея заключается в том, чтобы кортежи, срок хранения которых скоро истечет, а также кортежи, которые вряд ли привнесут много результатов в соединение, отбрасывать (если имеются ограничения на размер памяти) или вставлять в соединение, но игнорировать на шаге апробирования (если имеются ограничения по быстродействию процессора). Заметим, что возможны и другие цели, например получение случайной выборки из результата соединения.

В общем случае желательно ограничивать нагрузку таким образом, чтобы минимизировать снижение точности. Эта задача усложняется, когда имеется несколько запросов с большим количеством операторов, поскольку необходимо решить, в каком месте плана выполнения отбрасывать кортежи. Очевидно, что выгоднее отбрасывать кортежи как можно раньше, потому что тогда снижается нагрузка на все последующие операторы. Однако такая стратегия может негативно сказаться на точности многих запросов, если некоторые части плана общие. С другой стороны, если ограничивать нагрузку позже, после того как разделяемые подпланы уже вычислены, а остались только операторы, специфичные для отдельных запросов, то вряд ли получится сколько-нибудь заметно снизить общую нагрузку на систему.

В контексте ограничения нагрузки и генерирования плана запроса возникает вопрос: останется ли план, построенный без учета ограничения нагрузки, оптимальным и после ограничения. Показано, что это так для агрегирования по скользящему окну, но не для запросов с соединением скользящих окон.

Заметим, что вместо отбрасывания кортежей в периоды пиковой нагрузки можно откладывать их (например, сохранять на диске) и обрабатывать, когда пик утихнет. Наконец, отметим, что в случае периодического повторного выполнения постоянных запросов можно рассмотреть увеличение интервала между выполнениями как форму ограничения нагрузки.

10.3.2.3. Обработка не по порядку

До сих пор мы предполагали, что СПД обрабатывает входящие данные по порядку, обычно в порядке следования временных меток. Однако не всегда это возможно. Поскольку данные поступают из внешних источников, некоторые элементы могут запаздывать или приходить не в том порядке, в каком генерировались. Кроме того, от источника (например, удаленного датчика или маршрутизатора) может в течение некоторого времени вообще не приходить никаких данных, что может означать как отсутствие данных, так и выход источника из строя. В распределенных системах особенно это следует рассматривать как рутинные условия эксплуатации, связанные с разрывами сетевых соединений, длительностью восстановления и т. д. Поэтому нужно подумать о том, как обрабатывать данные, приходящие не по порядку.

Один из ранних подходов к решению этой проблемы состоял во введении «люфта», т. е. верхней границы количества пришедших не по порядку данных. Так, например, сделано в системе Aurora, где имеется буферизованный оператор сортировки, в котором данные накапливаются в течение времени люфта перед обработкой. Оператор выводит поток отсортированным по некоторому атрибуту. Данные, поступившие после истечения времени люфта, отбрасываются. В системе Truviso имеется понятие «дрейфа» на случай, когда потоки от одного источника данных сами по себе упорядочены, но данные от некоторых источников могут поступать с задержкой. Когда монитор замечает это, он запускает период дрейфа, в течение которого буферизует данные от других источников. Разница между этими решениями заключается в том, что в Aurora люфт можно задавать для каждого оператора, а в Truviso дрейфом управляет входной монитор.

Еще одно решение – использовать прерыватели, о которых мы уже упоминали. В данном случае прерывателем называется специальный кортеж, который содержит предикат, гарантированно истинный для последующей части потока данных. Например, прерыватель с предикатом `timestamp > 1262304000` гарантирует, что не поступит ни один кортеж с временной меткой меньшей, чем указанное время Unix; разумеется, если этот прерыватель генерируется источником, то он полезен, только если кортежи поступают в порядке временных меток. Прерыватели, которые налагают условия на временные метки будущих кортежей, обычно называют *пульсами* (heartbeat).

10.3.2.4. Многозапросная оптимизация

У запросов к базе данных могут быть одинаковые общие части, поэтому методы оптимизации пакета запросов давно вызывали интерес, есть даже специальное название – *многозапросная оптимизация*. В потоковых системах с поддержкой постоянных запросов имеется больше возможностей выявить общие компоненты и состояние. Например, агрегатные запросы к окнам разной ширины и, возможно, с разными значениями параметра `SLIDE` могут разделять общее состояние и структуры данных. Точно так же состояние и вычисления могут быть общими для похожих предикатов и соединений. Поэтому СПД могут группировать похожие запросы и выполнять один план запроса для всей группы.

На рис. 10.17 показаны некоторые проблемы, возникающие вследствие разделения планов запроса. Первые два плана соответствуют выполнению запросов Q_1 и Q_2 по отдельности, когда выборка производится раньше соединения. Третий план предназначен для выполнения обоих запросов, в нем сначала производится соединение, а затем выборка (заметим, что оператор соединения по существу создает две копии своего выходного потока). Несмотря на то что при выполнении обоих запросов часть работы разделяется, третий план может оказаться менее эффективным, чем индивидуальное выполнение, если лишь малая часть результата соединения удовлетворяет предикатам выборки σ_1, ..., σ_4. В таком случае оператор соединения выполняет много лишней работы. Четвертый план решает эту проблему, производя «предварительную фильтрацию» потоков перед соединением.

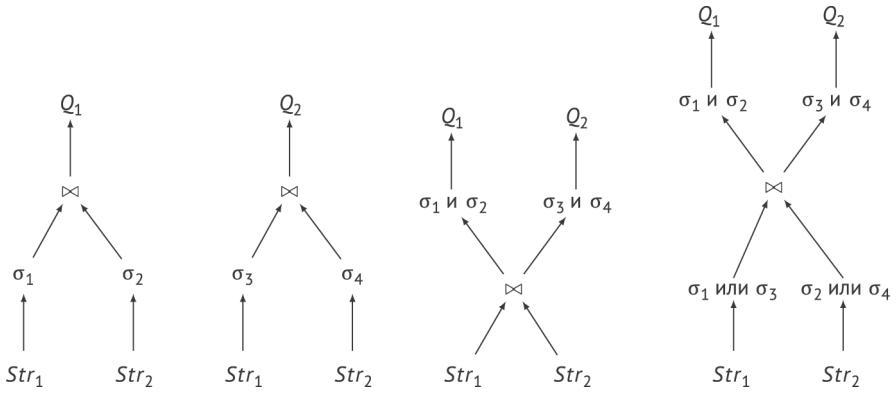


Рис. 10.17 ❖ Индивидуальные и разделяемые планы для запросов Q_1 и Q_2

10.3.2.5. Параллельная обработка потоков данных

Большая часть современных СПД работает в крупномасштабных кластерах, т. е. являются параллельными системами обработки потоковых данных (ПСОПД). Эти системы имеют много общих черт с параллельными базами данных, которые мы рассматривали в главе 8, и, что, пожалуй, важнее, с каркасами для обработки больших данных, описанными в разделе 10.2 этой главы. Поэтому далее мы будем опираться на этот материал и ссылаться на вышеупомянутые характеристики систем потоков данных.

Типичную среду выполнения в этих системах можно охарактеризовать как параллельное выполнение непрерывных операторов. Глядя на рис. 10.16, мы видим, что каждая вершина представляет собой отдельную операцию, назначенную нескольким узлам-исполнителям. Для простоты предположим, что каждый исполнитель выполняет только одну операцию. В этом контексте каждая машина-исполнитель выполняет назначенную ей операцию для одной секции потока данных и результаты потоком отправляет исполнителям, которые выполняют следующую операцию плана. Здесь важно отметить, что разбиение потока на секции происходит между каждой парой операций. Таким образом, выполнение операции состоит из трех шагов:

1) секционирование входного потока;
2) выполнение операции для одной секции;
3) агрегирование результатов от разных исполнителей (факультативно).

Секционирование потока

Как и во всех параллельных системах, цель секционирования – сбалансировать нагрузку на всех исполнителей, чтобы никто не отставал. Но здесь отличительная особенность состоит в том, что набор данных, назначаемый каждому исполнителю, поступает потоком, поэтому секционирование (по ключевому атрибуту) производится на лету, а не с помощью автономного процесса, как в системах, обсуждавшихся в разделе 10.2.

Простейший подход к балансировке нагрузки в распределенных системах – случайное распределение между исполнителями. *Секционирование та-*

сованием направляет элементы данных исполнителям циклически (поэтому его также называют *циклическим секционированием*). В результате нагрузка получается идеально сбалансированной. Для приложений, которые не хранят информацию о состоянии, этот метод работает отлично, но если состояние хранится, то нужно быть осторожнее. Поскольку элементы с одним и тем же ключом могут быть назначены разным исполнителям, для операций, имеющих состояние, необходим шаг агрегирования, на котором собираются частичные результаты каждого исполнителя для данного ключа (подробнее см. раздел 10.3.2.5). Агрегирование обходится дорого и должно учитываться. Кроме того, для операций, имеющих состояние, секционирование тасованием предъявляет высокие требования к памяти, т. к. каждый исполнитель должен хранить состояние для каждого ключа.

Другая крайность – секционирование хешированием, с этим методом мы уже встречались неоднократно. Хеширование гарантирует, что элементы данных с одинаковыми ключами будут назначены одному и тому же исполнителю, что позволяет отказаться от дорогостоящего шага агрегирования и минимизировать требования к памяти, потому что состояние для каждого ключа хранится только одним исполнителем. Однако нагрузка при этом может быть несбалансированной, особенно для асимметричных (с точки зрения распределения значения ключа) потоков.

Для приложений с запоминанием состояния секционирование тасованием дает верхнюю границу стоимости раздачи одинаковых ключей разным исполнителям, а секционирование хешированием – верхнюю границу несбалансированности нагрузки. Недавние работы сосредоточены на поиске алгоритмов секционирования между этими крайностями. Многообещающим является подход на основе расщепления ключей, когда после секционирования хешированием выполняется раздача одного ключа небольшому числу исполнителей, чтобы уменьшить дисбаланс. Цель состоит в том, чтобы сократить накладные расходы на агрегирование и одновременно добиться лучшей сбалансированности, особенно для асимметричных потоков данных. Алгоритм частичной группировки ключей (Partial Key Grouping – PKG) стремится уменьшить несбалансированность нагрузки при секционировании хешированием путем адаптации расщепления ключей. В PKG применяется принцип «выбора двух», когда разрешается расщеплять каждый ключ между двумя исполнителями. В результате PKG достигает гораздо лучшей сбалансированности нагрузки по сравнению с хеш-секционированием, ограничивая при этом коэффициент репликации и стоимость агрегирования. Для сильно несимметричных данных было предложено обобщение PKG, в котором можно выбирать не два, а больше исполнителей. Хотя показано, что сбалансированность нагрузки при этом улучшается, коэффициент репликации ограничен сверху количеством узлов-исполнителей в худшем случае. Еще один подход к обработке несимметричных данных – метод гибридного секционирования, когда кортежи в голове распределения значений ключей (частые) и в его хвосте (менее частые) обрабатываются по-разному, и предпочтение отдается хорошо сбалансированному назначению частых ключей, которые создают наибольшую нагрузку.

Выполнение операции для одной секции

Сначала сосредоточимся на выполнении отдельных операций. Для операций без состояния потоковое поступление данных не создает никаких особых проблем, и шаг агрегирования не нужен. Операции с запоминанием состояния требуют больше внимания, ими мы и займемся.

Если для операции с запоминанием состояния используется секционирование тасованием, то элементы данных с одинаковыми ключами могут быть размещены на разных исполнителях, каждый из которых будет хранить только частичные результаты. Поэтому для получения окончательного результата нужен шаг агрегирования. На рис. 10.18 показана операция подсчета, выполняемая тремя исполнителями; разными цветами обозначены различные ключи. Как видим, элементы данных с одинаковыми ключами могут попасть разным исполнителям, каждый из которых хранит счетчики вхождений каждого ключа (состояние), и в конце эти счетчики агрегируются.

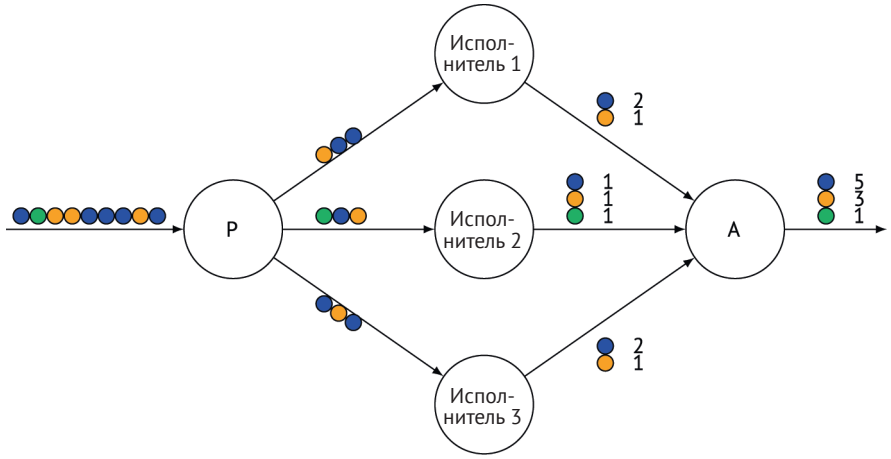


Рис. 10.18 ❖ Циклическое секционирование потока

Если для операции с запоминанием состояния используется секционирование хешированием, то все данные с одинаковыми ключами попадают одному исполнителю, поэтому шаг агрегирования не нужен. На рис. 10.19 показано секционирование хешированием для того же примера.

Если эти варианты включаются в план выполнения запроса, то каждая операция обычно выполняется отдельно, и решения о секционировании принимаются для каждой индивидуально, как в Apache Storm и Heron. Следовательно, план запроса на рис. 10.16 принимает вид, показанный на рис. 10.20.

Проблема возникает для сильно асимметричных потоков данных. В разделе 10.3.2.5 мы говорили, что в настоящее время общепринятым является подход на основе расщепления ключа с некоторыми оптимизациями на сильно асимметричных данных. Другой подход – повторное секционирование потока между операциями в плане запроса. Тогда фундаментальная проблема – как перенаправить поток данных между операциями в плане

запроса, для чего необходима также миграция состояния (поскольку часть потока поступит другому исполнителю). Было предложено несколько стратегий, но основополагающей работой на эту тему является идея Flux, которую мы возьмем в качестве примера при обсуждении повторного секционирования. Flux – это оператор потока данных, который помещается между двумя операциями в плане запроса; он отслеживает нагрузку на исполнителей и динамически перенаправляет потоки с одного исполнителя на другой, производя также миграцию состояния. Этот процесс состоит из двух этапов: перенаправление данных и миграция состояния. Для перенаправления нужно обновить внутренние таблицы маршрутизации. Миграция состояния

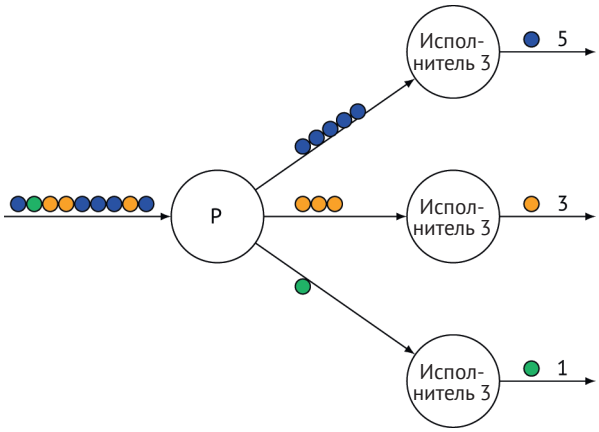


Рис. 10.19 ❖ Хеш-секционирование потока

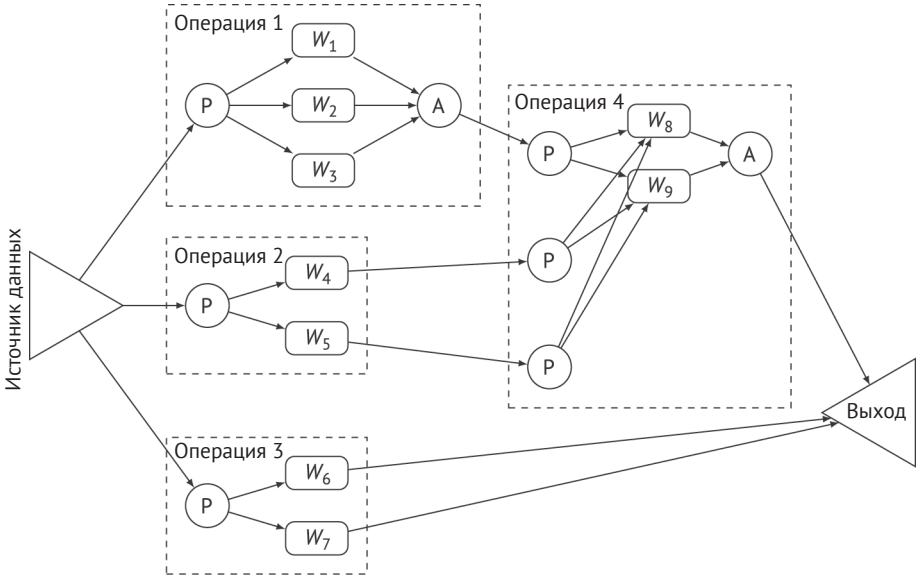


Рис. 10.20 ❖ Пример параллельного потокового плана запроса

сложнее, поскольку состояние, которое хранится на «старом» исполнителе, нужно сериализовать и переместить на «новый». Это включает следующие шаги: остановить поступление новых кортежей в старую секцию; сериализовать состояние, хранящееся на «старом» исполнителе, для чего нужно извлечь информацию из внутренних структур данных; переместить состояние на «новый» исполнитель; десериализовать состояние и заполнить соответствующие структуры данных на «новой» машине; возобновить поступление данных из потока. Очевидно, что миграция состояния должна быть быстрой, но это непростая задача, в которой задействованы протоколы синхронизации. Именно по этой причине современные системы обычно не включают встроенной поддержки миграции состояния.

10.3.3. Отказоустойчивость СПД

Проблематика надежности распределенных/параллельных СПД и реляционных СУБД в чем-то схожа, но возникают дополнительные трудности, связанные с необходимостью обрабатывать потоки, протекающие через планы выполнения. Сначала вернемся к упомянутому ранее различию между системами, в которых разбивается на части план выполнения и каждая часть выполняется в отдельном узле, и системами, в которых секционируются данные, а план выполнения реплицируется в каждом узле (распараллеливание по данным). Во втором случае отказы можно обработать, применяя методы репликации исполнителей, которые мы обсуждали в разделе 10.2.1. Но в первом случае серверы «связаны», поскольку выполняют части плана запроса, и данные текут от начальных серверов к конечным. Поэтому отказ узла может оборвать выполнение запроса из-за потери важной (промежуточной) информации о состоянии и остановки расположенных ниже по потоку серверов, которые перестали получать данные. Следовательно, в таких системах нужны действенные средства обеспечения доступности.

Важный вопрос – поддерживаемая системой семантика выполнения запроса, когда поток данных протекает по сети серверов (или через план запроса). Есть три варианта: не менее одного раза, не более одного раза и ровно один раз. Семантика «*не менее одного раза*» (или *восстановление с откатом*) означает, что система гарантирует, что каждый элемент данных будет обработан хотя бы один раз, но не дает никаких гарантий относительно дубликатов. Таким образом, если элемент данных снова будет направлен вышедшим из строя узлом после восстановления, то этот элемент может быть обработан еще раз, что приведет к дубликату на выходе. С другой стороны, если принята семантика «*не более одного раза*» (или *восстановление с пропуском*), то некоторые данные могут быть не обработаны вовсе, зато система гарантирует, что дубликаты будут обнаружены и отброшены. Это может быть результатом политики ограничения нагрузки (см. выше) или отказа узла, который после восстановления игнорирует все данные, которые мог бы получить за время простоя. Наконец, семантика «*ровно один раз*» (или *точное восстановление*) означает, что система обрабатывает каждый элемент данных строго один раз – ничего не пропадает и не обрабатывается повторно. Разумеется, для

каждого случая нужна своя функциональность системы. Все три семантики встречаются на практике: Apache Storm и Heron предоставляют приложениям выбор между семантикой «не менее одного раза» и «не более одного раза», а Spark Streaming, Apache Flink и MillWheel поддерживают семантику «ровно один раз».

Методы восстановления СПД можно разбить на два основных класса: репликация и вышестоящая копия. В случае репликации для каждого узла, выполняющего часть плана запроса, существует узел реплики, также отвечающий за эту часть плана. Это конфигурация первичный–вторичный, в которой первичный узел обслуживает план запроса, пока исправен, а вторичный подхватывает работу, если первичный выходит из строя. При этом вторичный узел может быть настроен как активный резерв – оба узла получают данные от вышестоящих узлов и оба обрабатывают их одновременно, но только первичный посылает результаты дальше, – или как пассивный резерв, когда первичный периодически посылает дельту своего состояния вторичному, а вторичный, соответственно, обновляет собственное состояние. Оба варианта можно подкрепить записью контрольных точек, чтобы восстановление происходило быстрее. Такой подход был предложен как часть вышеупомянутого оператора Flux и использован в системе Borealis. Альтернатива – *вышестоящая копия*: когда вышестоящие узлы буферизуют элементы данных, отправляемые нижестоящим, до тех пор пока те не будут обработаны. Если нижестоящий узел выйдет из строя, а затем восстановится, то он получит данные от вышестоящих узлов и обработает их повторно. Проблема заключается в том, чтобы решить, насколько велики должны быть буферы, чтобы в них поместились данные, поступившие за время отказа и восстановления. Она дополнительно осложняется тем, что на решение влияет темп поступления данных и другие факторы. Такой подход принят в Apache Storm и TimeStream.

10.4. Платформы для анализа графов

Данные, представленные в виде графов, играют все большую роль во многих приложениях. В этом разделе мы обсудим *графы свойств*, в которых с вершинами и ребрами ассоциированы атрибуты. Еще один тип графов – каркас описания ресурсов (Resource Description Framework – RDF) – мы рассмотрим в главе 12. Графы свойств применяются для моделирования сущностей и связей в различных дисциплинах: биоинформатике, программной инженерии, электронной торговле, финансах, биржевых торгах и социальных сетях. Граф $G = (V, E, D_V, D_E)$ представляет собой множество вершин V и множество ребер E[1], где D_V и D_E определены ниже. Граф свойств имеет следующие отличительные особенности:

- каждая вершина графа представляет некоторую сущность, а ребро между двумя вершинами – связь между соответствующими сущностями.

[1] В этом разделе мы будем обозначать V_G и E_G множества вершин и ребер графа G, но если очевидно, о каком графе идет речь, то нижние индексы будем опускать.

Например, в графе социальной сети Facebook каждая вершина представляет пользователя, а ребро – отношение «дружбы» между двумя пользователями;

- две вершины могут быть соединены несколькими ребрами, каждое из которых представляет отдельную связь; такие структуры обычно называют *мультиграфами*;
- с каждым ребром может быть ассоциирован вес (*взвешенные графы*), его семантика зависит от конкретного графа;
- графы могут быть *ориентированными* и *неориентированными*. Например, граф Facebook обычно неориентированный, поскольку описывает симметричное отношение между двумя пользователями: если пользователь A является другом B, то B является другом A. Напротив, граф сети Twitter, в котором ребра представляют отношение «наблюдает» (follows), является ориентированным, поскольку из того, что пользователь A наблюдает[1] за B, необязательно следует, что B наблюдает за A. Напомним, что RDF-графы, по определению, ориентированные;
- с каждой вершиной и каждым ребром может быть ассоциировано множество атрибутов (свойств), описывающих свойства сущности (в случае вершины) или связи (в случае ребра). Если свойства ассоциированы с ребрами, то граф называется *реберно-помеченным*. D_V и D_E в определении графа – это множества свойств вершин и ребер соответственно. У каждой вершины (ребра) может быть свой набор свойств, в общем случае, говоря о свойствах графа, мы будем писать D или D_G.

Реальные графы, являющиеся предметом анализа (например, графы социальных сетей, дорожной сети или веба), обладают рядом характеристик, оказывающих существенное влияние на проектирование системы:

1) эти графы очень велики, иногда насчитывают миллиарды вершин и ребер. Обработка графов с таким числом вершин, и особенно ребер, требует аккуратности;
2) многие графы такого рода называют степенными[2], или безмасштабными, поскольку степени вершин изменяются в очень широких пределах (это явление называется *асимметрией распределения вершин*). Например, средняя степень вершин в графе Twitter равна 35, но существуют «суперузлы», степень которых достигает 2,9 млн[3];
3) во многих реальных графах средняя степень вершин очень велика и имеются плотные ядра. Например, средняя степень вершин в графе сети Friendster равна 55, а в графе Facebook – 190;
4) у некоторых реальных графов очень большой диаметр (количество переходов между двумя вершинами, удаленными друг от друга на наи-

---

[1] Специально для поклонников нерусского жаргона – «является фолловером». – *Прим. перев.*

[2] От «степенной закон» (power law), а не от «степень» вершины (см. https://en.wikipedia.org/wiki/Power_law). – *Прим. перев.*

[3] Предупреждаем, что графы со временем эволюционируют, и эти значения изменяются. Их следует рассматривать как указание на порядок величины, а не точное значение.

большее расстояние). К таковым относятся пространственные графы (например, графы дорожной сети) и веб-графы: диаметр веб-графов достигает нескольких сотен, а в некоторых дорожных сетях он гораздо больше. Диаметр графа влияет на алгоритмы анализа, в которых требуется посетить каждую вершину и выполнить в ней какие-то вычисления (мы обсудим этот вопрос ниже).

Эффективная обработка таких графов – существенная часть платформ больших данных. Как часто бывает с большими данными, большинство таких платформ параллельные или распределенные (горизонтально масштабируемые), т. е. граф размещается в узлах кластера или распределенной системы.

Задачи, решаемые на графах, можно отнести к двум классам. Первый – *аналитические запросы* (или *аналитические рабочие нагрузки*), когда вычисление состоит в итеративной обработке всех вершин графа, пока не будет достигнута сходимость. Примерами могут служить вычисление PageRank (см. пример 10.4), кластеризация, нахождение компонент связности (см. пример 10.5) и многие алгоритмы машинного обучения, в которых используются графовые данные (например, распространение доверия). Мы рассмотрим различные вычислительные подходы, разработанные для таких задач, а также системы, которые их поддерживают. Существуют специализированные платформы итеративных вычислений, о которых мы упоминали при обсуждении Spark в предыдущем разделе. В этом разделе мы остановимся на них подробнее. Второй класс задач – *онлайновые запросы* (или *онлайновые рабочие нагрузки*). Они не являются итеративными и обычно нуждаются в доступе к части графа, при выполнении могут помочь специальные вспомогательные структуры данных, например индексы. Назовем несколько примеров онлайновых запросов: запросы о достижимости (достижима ли конечная вершина из начальной), запросы о кратчайшем пути между двумя вершинами и сопоставление подграфов (задача об изоморфизме графов). Мы отложим обсуждение этих рабочих нагрузок до главы 11, посвященной графовым СУБД.

Пример 10.4. PageRank – хорошо известный алгоритм для вычисления важности веб-страниц. Его идея заключается в том, что важность страницы определяется количеством и качеством ссылок на нее с других страниц. Качеством в данном случае является ранг страницы PageRank (поэтому определение рекурсивно). Каждая веб-страница представлена вершиной в веб-графе (см. рис. 10.21), а каждое ориентированное ребро представляет отношение «указывает на». Таким образом, PageRank веб-страницы P_i, обозначаемый $PR(P_i)$, равен сумме PageRank'ов всех указывающих на нее страниц P_j, поделенной на количество страниц, на которые указывает P_j. Идея в том, что если страница P_i указывает на n страниц (одной из которых является P_i), то ее PageRank вносит равный вклад в вычисление PageRank'ов n страниц. В формуле PageRank имеется также поправочный коэффициент, основанный на теории случайного блуждания: если пользователь начинает с некоторой веб-страницы и переходит по ссылкам на другие веб-страницы, то такое «блуждание» рано или поздно остановится. Находясь на странице P_i, пользователь с вероятностью d продолжает щелкать по ссылкам и с вероятностью $1 - d$ останавливается; в результате эмпирических исследований установлено, что d равна 0.85. Поэтому если обозначить B_{P_i} множество входящих соседей P_i (страниц, которые

ссылаются на P_i), а F_{P_i} – множество исходящих соседей P_i (страниц, на которые ссылается P_i), то формула PageRank имеет вид

$$PR(P_I) = (1-d) + d \sum_{P_j \in B_{P_i}} \frac{PR(P_j)}{|F_{P_j}|}.$$

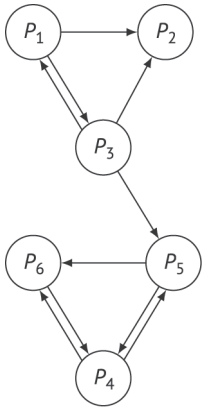


Рис. 10.21 ❖ Представление веб-графа для вычисления PageRank

Более подробно мы обсудим PageRank в главе 12, когда будем говорить об управлении веб-данными. А пока просто сосредоточимся на его вычислении для примера на рис. 10.21. Рассмотрим страницу P_2; PageRank этой страницы равен $PR(P_2) = 0.15 + 0.85(PR(P_1)/2 + PR(P_3)/3)$. Очевидно, что это рекурсивная формула, потому что в ней участвуют значения PageRank для P_1 и P_3. Обычно вычисление начинается с назначения всем вершинам одинаковых значений PageRank (в данном случае 1/6, потому что всего имеется 6 вершин), после чего производится несколько итераций вычисления значения PageRank всех вершин, пока не наступит сходимость (т. е. значения перестанут изменяться). Таким образом, вычисление PageRank обладает обоими свойствами аналитических рабочих нагрузок: оно итеративное и в каждой итерации участвуют все вершины[1].

Пример 10.5. В качестве второго примера рассмотрим вычисление компонент связности графа. Сначала несколько вступительных слов. Говорят, что граф *связный*, если между любыми двумя его вершинами существует путь. Максимальный связный подграф графа называется *компонентой связности* – каждая вершина такой компоненты достижима из любой другой ее вершины. Нахождение компонент связности графа – важная аналитическая задача, которая находит применение в ряде приложений, например кластеризации. Если граф ориентированный, то всякий подграф, в котором между любыми

[1] Разработаны различные варианты вычисления PageRank, но в этом обсуждении мы их касаться не будем.

двумя вершинами существуют ориентированные пути в обоих направлениях (т. е. путь из *v* и *u* и путь из *u* в *v*), называется *компонентой сильной связности*. Например, на рис. 10.21 $\{P_1, P_3\}$ и $\{P_4, P_5, P_6\}$ – компоненты сильной связности. Если все ориентированные в этом графе заменить неориентированными, а затем найти максимальные компоненты связности, то получим множество компонент слабой связности. На рис. 10.21 весь граф является единственной компонентой слабой связности (в таком случае говорят, что граф *слабо связный*).

Для нахождения компонент слабой связности применяется итеративный алгоритм с использованием поиска в глубину (depth first search – DFS). Если имеется граф $G = (V, E)$, то для нахождения компоненты, которой принадлежит вершина $v \in V$, следует выполнить поиск в глубину, начиная с v. ♦

10.4.1. Разбиение графа

Как уже было сказано, большинство систем для анализа графов параллельные, поэтому граф необходимо разбить на части и назначить каждой части узел-исполнитель. В главе 2 мы уже имели дело с секционированием данных в контексте распределенных реляционных систем, а в главе 8 – в контексте параллельных систем баз данных. Секционирование, или, как обычно говорят, разбиение графов, устроено по-другому из-за того, что вершины соединены между собой; в каком-то смысле это похоже на межфрагментные ограничения целостности в распределенных системах, но с графами дело обстоит сложнее из-за высокой степени взаимодействия между вершинами, как станет ясно в последующих разделах. Поэтому для разбиения графов разработаны специальные алгоритмы, и на эту тему существует весьма обширная литература.

К разбиению графов можно подойти двумя способами: *реберное разрезание* (*вершинно-непересекающееся* разбиение) или *вершинное разрезание* (*реберно-непересекающееся* разбиение). В первом случае каждая вершина попадает только в один раздел, но ребра, соединяющие пограничные вершины, реплицируются в обоих разделах. Во втором случае каждое ребро попадает только в один раздел, но вершины, инцидентные ребрам, оказавшимся в разных разделах, реплицируются. В обоих случаях преследуются три цели: (1) распределить вершины или ребра по разделам, так чтобы разделы не пересекались; (2) обеспечить сбалансированность разделов; (3) минимизировать количество разрезов (реберных или вершинных), так чтобы взаимодействие между машинами, на которых размещены разные разделы, было сведено к минимуму. Найти компромисс между этими требованиям – трудная задача; например, если бы нас интересовала только сбалансированность рабочей нагрузки при соблюдении требования об отсутствии пересечений, то достаточно было бы распределять вершины (или ребра) циклически. Но нет гарантии, что при этом не получится слишком много разрезов.

Задачу о разбиении можно сформулировать как задачу оптимизации. Пусть дан граф $G(V, E)$ (о свойствах на время забудем). Требуется найти разбиение $P = \{P_1, \ldots, P_k\}$ графа G на k разделов со сбалансированными размерами P_i такое, что

$C(P)$ достигает минимума

при условиях

$$w(P_i) \leq \beta * \frac{\sum_{j=1}^{k} w(P_j)}{k}, \quad \forall i \in \{1 \ldots k\},$$

где $C(P)$ – полная стоимость коммуникации при таком разбиении, а $w(P_i)$ – абстрактные накладные расходы на обработку раздела P_i. Два рассмотренных выше подхода (вершинное и реберное разрезания) отличаются определениями $C(P)$ и $w(P_i)$, о чем мы скажем ниже. В этой постановке β – параметр, разрешающий неточную сбалансированность разбиения. При β = 1 решением является точно сбалансированное разбиение, а задача известна под названием оптимизация k-сбалансированного разбиения графа. При β > 1 допускается отклонение от точной сбалансированности, и задача называется оптимизацией (k, β)-сбалансированного разбиения графа. Доказано, что эта задача NP-трудная, поэтому предложены различные эвристики для нахождения приближенного решения.

При реберном (вершинно-непересекающемся) разрезании эвристический подход заключается в том, чтобы попытаться найти сбалансированное распределение вершин по разделам, минимизирующее количество реберных разрезов. Поэтому каждый раздел P_i представляет собой множество вершин, и $w(P_i)$ определяется в терминах числа вершин в разделе (т. е. $w(P_i) = |P_i|$), тогда как стоимость коммуникации вычисляется как доля реберных разрезов:

$$C(P) = \frac{\sum_{i=1}^{k} |e(P_i, V \setminus P_i)|}{|E|},$$

где $|e(P_i, P_j)|$ – количество ребер между разделами P_i и P_j.

Самый известный эвристический алгоритм вершинно-непересекающегося разрезания, METIS, находит разбиение, близкое к оптимальному. Он состоит из трех шагов.

1. Для данного графа $G_0 = (V, E)$ строится иерархия все более огрубленных графов $G_1, \ldots, G_n$ такая, что $|V(G_i)| > |V(G_j)|$ для $i < j$. Существует много разных способов огрубления, но самый популярный, называемый *уплотнением*, заключается в замене множества вершин G_i одной вершиной G_j ($i < j$). Огрубление обычно прекращается, когда G_n настолько мал, что уже можно применить более дорогостоящий алгоритм разбиения. Граф G_i преобразуется в G_{i+1} путем нахождения *максимального паросочетания*, т. е. множества ребер, никакие два из которых не имеют общей вершины. Затем концы этих ребер представляются одной вершиной G_{i+1}.
2. G_n разбивается с помощью какого-то алгоритма разбиения – как уже отмечалось, к этому моменту G_n должен быть достаточно мал, чтобы можно было воспользоваться любым предпочтительным алгоритмом разбиения, не обращая внимания на стоимость вычислений.
3. G_n итеративно восстанавливается в G_0, при этом на каждом шаге:
   a) решение о разбиении графа G_j проецируется на граф G_{j-1} (заметим, что чем меньше индекс, тем детальнее граф) и
   b) разбиение G_{j-1} улучшается с применением различных методов.

Хотя METIS и другие подобные алгоритмы значительно уменьшают время анализа графов, они практически неприменимы даже для графов среднего размера из-за высокой стоимости вычислений. Затраты на разбиение – важный фактор в любой системе анализа графов, поскольку время загрузки и разбиения может составлять значительную долю всего времени обработки.

Простой эвристический алгоритм вершинно-непересекающегося разбиения, основанный на хешировании, включен в состав большинства обсуждаемых ниже систем анализа графов. В этом случае вершина назначается разделу, определяемому хеш-значением ее идентификатора. Это просто, очень быстро и могло бы хорошо балансировать нагрузку в графах с равномерным распределением степеней вершин. Однако если распределение степеней несимметрично, как часто бывает в реальной жизни, то в результате можно получить несбалансированную нагрузку. В моделях реберного разрезания нагрузка распределяется в терминах вершин, но для некоторых алгоритмов нагрузка пропорциональна количеству ребер, поэтому для асимметричных графов окажется неравномерной. Для таких случаев разработаны более сложные эвристики, учитывающие структуру графа.

В одном из таких подходов – *распространении меток* – каждой вершине в начале работы назначается уникальная метка, а затем вершины итеративно обмениваются метками с соседями. На каждой итерации каждая вершина получает метку, чаще всего встречающуюся по «соседству»; если у нескольких меток частоты одинаковы, для выбора применяется специальный метод. Этот процесс останавливается, когда метки перестают изменяться. Алгоритм чувствителен к структуре графа, но не гарантируется, что он создаст сбалансированное разбиение. Один из способов добиться сбалансированности – начать с несбалансированного разбиения, а затем использовать жадный алгоритм распространения меток, чтобы переместить вершины между разделами, получив таким образом сбалансированное (или почти сбалансированное) разбиение. Жадный алгоритм перемещает вершины, стремясь максимизировать функцию полезности перемещения при ограничениях на сбалансированность. В качестве функции полезности можно взять, например, количество соседних вершин, которые окажутся в одном разделе. Есть возможность скомбинировать METIS с алгоритмом распространения меток, включив последний на этапе огрубления. И на этот раз ставится задача ограниченной максимизации функции полезности, учитывающей соседство вершин, с целью свести к минимуму количество реберных разрезов.

Пример 10.6. Рассмотрим граф на рис. 10.22a. Его вершинно-непересекающееся разбиение показано на рис. 10.22b, где реберные разрезы обозначены штриховыми линиями. Это разбиение было получено с помощью описанного выше хеширования. Заметим, что в результате придется разрезать 10 из 12 ребер. Этот пример демонстрирует трудность задачи разрезания графа с вершинами большой степени (в этом графе велика степень вершины v_3 и особенно v_4), при котором получается много реберных разрезов. METIS справляется этим графом лучше, но вместо трех разделов порождает всего два: $\{v_1, v_3, v_4, v_7, v_9\}$ и $\{v_2, v_5, v_6, v_8\}$, для чего хватает пяти разрезов (разбиение на две части, основанное на хешировании, дает 8 разрезов). ♦

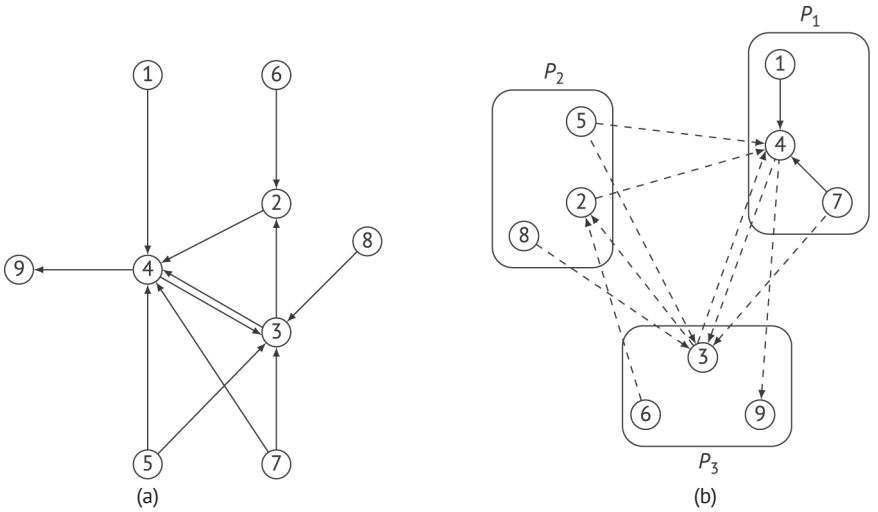


Рис. 10.22 ❖ Пример разбиения:
(a) исходный граф; (b) вершинно-непересекающееся (реберное) разбиение

Доказано, что эвристика реберных разрезов хорошо работает для графов с вершинами небольшой степени, но плохо для степенных графов – при этом количество реберных разрезов велико. Алгоритм METIS был модифицирован для решения этой конкретной проблемы, но его производительность при работе с очень большими графами остается проблемой. Принято считать, что алгоритмы вершинного разрезания, которые распределяют ребра по разделам (P_i содержат непересекающиеся множества ребер), реплицируя вершины, инцидентные этим ребрам, легче справляются со степенными графами. В этом случае $w(P_i)$ определяется как количество ребер в разделе P_i, т. е. $w(P_i) = |e(P_i)|$. Для таких эвристик на стоимость коммуникации $C(P)$ влияет коэффициент репликации каждой вершины (т. е. количество разделов, к которым эта вершина отнесена); ее можно записать в виде

$$C(P) = \frac{\sum_{v \in V} |A(v)|}{|V|},$$

где $A(v) \subseteq \{P_1, \ldots, P_k\}$ представляет множество разделов, к которым отнесена вершина v.

Хеширование можно применить и к вершинному разрезанию: в этом случае хешируются идентификаторы обеих вершин, инцидентных ребру. Это простой и быстрый (поскольку легко поддается распараллеливанию) метод, который должен давать хорошо сбалансированное разбиение. Однако он может приводить к слишком сильной репликации вершин. Можно, впрочем, использовать хеширование, контролируя при этом коэффициент репликации. Один из предложенных подходов состоит в том, чтобы для ребра $e_{u,v}$ определить *множества ограничений* C_u и C_v для инцидентных ему вершин u и v, т. е. множества разделов, на которые можно реплицировать u и v. Очевидно, что ребро $e_{u,v}$ должно быть отнесено к разделу, общему для множеств ограничений u и v, т. е. к пересечению $C_u \cap C_v$. Множества ограничений ограничивают

количество разделов, к которым может быть отнесена вершина, и тем самым контролируют верхнюю границу коэффициента репликации. Чтобы сгенерировать множества ограничений, можно определить квадратную матрицу разделов, хешировать u (и аналогично v) в один из разделов (скажем, P_i) и взять в качестве C_u (соответственно C_v) разделы, которые находятся в той же строке или том же столбце, что P_i.

Была также предложена эвристика вершинного разрезания, принимающая во внимание особенности графа. Жадный алгоритм решает, в какой раздел поместить $(I + 1)$-е ребро, чтобы минимизировать коэффициент репликации. Конечно, размещение $(I + 1)$-го ребра зависит от размещения первых I ребер, поэтому история размещений важна. Местоположение ребра $e_{u,v}$ определяется следующими правилами:

1) если пересечение $A(u)$ и $A(v)$ не пусто (т. е. существуют разделы, содержащие одновременно u и v), то отнести $e_{u,v}$ к одному из разделов, принадлежащих пересечению;
2) если пересечение $A(u)$ и $A(v)$ пусто, но $A(u)$ и $A(v)$ по отдельности не пусты, то отнести $e_{u,v}$ к тому из разделов, принадлежащих $A(u) \cup A(v)$, в котором больше всего еще не размещенных ребер;
3) если только одно из множеств $A(u)$ и $A(v)$ не пусто (т. е. только одна из вершин u и v отнесена к разделам), то отнести $e_{u,v}$ к одному из разделов, в которых находится уже размещенная вершина;
4) если и $A(u)$, и $A(v)$ пусты, отнести $e_{u,v}$ к наименьшему разделу.

Этот алгоритм учитывает структуру графа, но его трудно распараллелить для достижения высокой производительности, поскольку он зависит от истории. Для распараллеливания необходимо хранить и периодически обновлять глобальное состояние или согласиться на приближение, когда каждая машина рассматривает только свою собственную локальную историю, игнорируя глобальную.

Можно также сочетать реберное и вершинное разрезания в одном алгоритме разбиения. Например, в системе PowerLyra реберное разрезание применяется для вершин малой степени, а вершинное – для вершин большой степени. Точнее, ориентированное ребро $e_{u,v}$ хешируется по v, если степень v мала, и по u, если степень велика.

Пример 10.7. Снова рассмотрим граф на рис. 10.22а. Реберно-непересекающееся разбиение этого графа показано на рис. 10.23, где реплицированные вершины изображены штриховыми окружностями. Это разбиение было получено, как описано выше. Заметим, что реплицировано 6 из 9 вершин. ♦

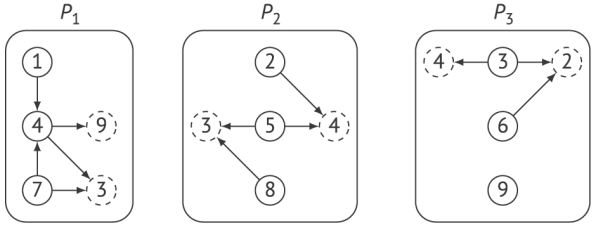


Рис. 10.23 ❖ Реберно-непересекающееся разбиение (вершинное разрезание)

10.4.2. MapReduce и анализ графов

Реализацию MapReduce, например Hadoop, можно использовать для обработки и анализа графов. Однако в разделе 10.2 мы отмечали, что системы MapReduce плохо приспособлены к итеративным вычислениям, и там же обсудили основные возникающие при этом проблемы. В графовых системах имеется дополнительная проблема несбалансированного назначения вершин исполнителям, связанная с тем, что во многих реальных графах распределение степеней вершин далеко от равномерного (см. раздел 10.4.1), поэтому затраты на коммуникацию в разных узлах-исполнителях сильно различаются. Все это влечет за собой значительные издержки, которые неблагоприятно сказываются на производительности MapReduce в применении к анализу графов. Однако в большинстве специализированных систем анализа графов, обсуждаемых в следующем разделе, требуется, чтобы весь граф целиком помещался в памяти; если это невозможно, то каркас MapReduce мог бы стать разумной альтернативой, и в некоторых исследованиях изучается его применение для различных рабочих нагрузок, а также модификации, позволяющие улучшить масштабируемость. Существуют также системы, в которых MapReduce модифицирован, чтобы полнее соответствовать задачам анализа графов. Как уже отмечалось, Spark является развитием MapReduce для итеративной обработки, а поверх Spark разработана система GraphX для обработки графов. Еще один вариант MapReduce для работы с графами – система HaLoop. В обоих случаях состояние, изменяющееся от итерации к итерации, отделено от инвариантных данных, которые кешируются во избежание лишнего ввода-вывода. Также модифицирован планировщик, который гарантирует, что одни и те же данные попадают одним и тем же узлам на разных итерациях. Реализации, конечно, различаются. В HaLoop изменен планировщик задач Hadoop в мастер-узле и отслеживатель заданий в узлах-исполнителях, а также реализован контроль циклов в мастер-узле для проверки сходимости. С другой стороны, в GraphX используются модификации, уже включенные в Spark для более эффективного решения итеративных задач. В этой системе строится реберно-непересекающееся разбиение графа и в каждом узле-исполнителе создаются таблицы вершин и ребер. В каждой записи таблицы вершин хранится идентификатор и свойства вершины, а в каждой записи таблицы ребер – концевые точки и свойства ребра. Эти таблицы реализованы в виде RDD-наборов Spark. Всякое вычисление на графе состоит из двух итеративно повторяемых шагов: соединить таблицы вершин и ребер и выполнить агрегирование. Для соединения необходимо переместить таблицы вершин на узлы-исполнители, в которых хранятся соответствующие таблиц ребер, поскольку число вершин меньше числа ребер. Чтобы не рассылать каждую таблицу вершин во все узлы с таблицами ребер, GraphX создает таблицу маршрутизации, в которой для каждой вершины перечислены исполнители, хранящие таблицы ребер, в которых эта вершина упоминается; она также реализована в виде RDD.

10.4.3. Специализированные системы анализа графов

Теперь обратимся к системам, специально разработанным для анализа графов. В основу классификации можно положить *модель программирования* или *модель вычислений*. Модель программирования определяет, как программист должен писать алгоритмы, выполняемые системой, а модель вычислений – как система исполняет эти алгоритмы.

Существует три основные модели программирования: ориентированная на вершины, на разделы и на ребра.

- **Модель, ориентированная на вершины.** В такой модели программист должен описывать вычисления, выполняемые в каждой вершине. Поэтому ее часто называют «думай как вершина». Вычисления в вершине *v* основаны только на ее собственном состоянии и на состояниях соседних вершин. Например, при вычислении PageRank каждая вершина программируется для получения результатов вычисления ранга от соседей и вычисления своего ранга на их основе. Затем результаты вычисления состояния становятся доступны соседним вершинам, которые могут выполнить свои вычисления.
- **Модель, ориентированная на разделы.** В системах, следующих этой модели, программист описывает вычисления, выполняемые для раздела в целом, а не для каждой вершины. Иногда такую модель называют *блочно-центрической*, поскольку вычисления производятся над блоком вершин. Встречаются также названия «думай как блок» или «думай как граф».

  В ориентированной на разделы модели обычно используется последовательный алгоритм внутри каждого блока, которому необходимы только состояния соседних блоков целиком, а не состояния отдельных вершин. Наличие разных алгоритмов вычислений внутри раздела и между разделами (для пограничных вершин) может усложнить логику, но уменьшает зависимость от состояния соседей и накладные расходы на коммуникацию.
- **Модель, ориентированная на ребра.** Третий подход двойствен модели, ориентированной на вершины, поскольку описываются операции для каждого ребра, а не для каждой вершины. В этом случае в фокусе внимания находится ребро, а не вершина, поэтому можно было бы назвать такую модель «думай как вершина».

Существует три модели вычислений: синхронная пошагово-параллельная (bulk synchronous parallel – BSP), асинхронная параллельная (asynchronous parallel – AP) и сбор–обработка–распространение (gather-apply-scatter – GAS).

- **Синхронная пошагово-параллельная.** Модель BSP – это модель параллельных вычислений, в которой вычисление разбито на несколько супершагов, разделенных глобальными барьерами. На каждом супершаге все обрабатывающие узлы (т. е. исполнители) параллельно

выполняют некоторое вычисление, а в конце супершага все они синхронизируются, перед тем как перейти к следующему супершагу. Синхронизация сводится к разделению состояния, вычисленного на этом супершаге, со всеми остальными исполнителями, чтобы на следующем супершаге все могли им пользоваться. Супершаги продолжаются, пока не будет выполнено условие сходимости. На рис. 10.24 показан пример BSP-вычисления в трех узлах, которое сходится после трех шагов.

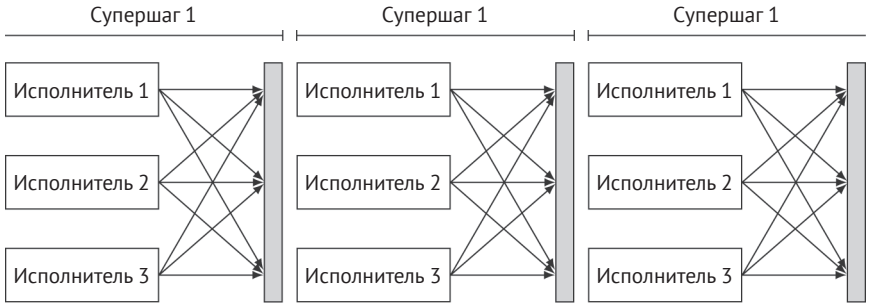


Рис. 10.24 ❖ Модель вычислений BSP

Поскольку большинство таких систем работают в параллельных кластерах, взаимодействие обычно принимает вид передачи сообщений (это верно для всех обсуждаемых нами моделей вычислений). В модели BSP реализовано взаимодействие по типу проталкивания: сообщения посылаются отправителем без запроса и буферизуются получателем. Прием сообщения в конце супершага автоматически подготавливает получателя к участию в следующем супершаге. Модель BSP упрощает параллельные вычисление, но нуждается в тщательном планировании разбиения на задачи, чтобы нагрузка на все машины-исполнители была в разумной степени сбалансирована и не возникало отстающих. Кроме того, приходится нести затраты на синхронизацию в конце каждого супершага.

- **Асинхронная параллельная.** В модели AP снято ограничение модели BSP – требование синхронизации исполнителей при достижении глобального барьера: состояния, вычисленные на супершаге k, могут быть использованы на супершаге $k + 1$, даже если доходят до узла назначения внутри супершага k. В модели AP глобальные барьеры между супершагами сохранены, но полученное состояние видно и может использоваться немедленно. Поэтому вычисление на супершаге k может быть основано на состояниях соседей, которые были вычислены на супершаге $k - 1$, но отложены и не получены до конца супершага $k - 1$ или на супершаге k.

  Тот факт, что вычисление состояния на одном супершаге может перекрываться с получением состояния от соседей, порождает проблемы согласованности: изменением и чтением состояния нужно аккуратно управлять. Обычно для этой цели на состояния ставятся блокировки

на время их чтения или записи, а поскольку состояния распределены по нескольким узлам, необходимо применять методы распределенной блокировки.

Важная цель модели AP – улучшить производительность, дав более быстрым узлам продолжать обработку, не стоя в ожидании у барьера синхронизации[1]. Количество супершагов может при этом уменьшиться. Но поскольку барьеры синхронизации все-таки остаются, накладные расходы на синхронизацию и коммуникацию полностью не устранены.

- **Сбор–обработка–распространение.** Как следует из самого названия, модель сбора–обработки–распространения (GAS) состоит из трех этапов. На этапе сбора элемент графа (вершина, блок или ребро) получает (или вытягивает) информацию о своих соседях, на этапе обработки он использует собранные данные для вычисления собственного состояния, а на этапе распространения обновляет состояние соседей. Важная отличительная особенность GAS – отделение обновления состояния от активации. В моделях BSP и AP соседи, получив обновление состояния, автоматически активируются (т. е. планируется их выполнение), тогда как в GAS эти два действия разделены. Это разделение важно, потому что позволяет планировщику самостоятельно решать, какие элементы графа выполнять дальше (быть может, принимая во внимание приоритеты).

  Модель GAS может быть синхронной и асинхронной. Синхронная GAS похожа на модель BSP тем, что глобальные барьеры существуют, но есть одно существенное отличие: в BSP каждый элемент графа проталкивает свое состояние соседям в конце супершага, тогда как в GAS активированный элемент графа вытягивает состояние своих соседей в начале супершага.

  В асинхронной GAS глобальных барьеров нет, поэтому возникает та же проблема согласованности, что в модели AP, и решается она с помощью распределенных блокировок. Однако от модели AP она отличается отсутствием супершагов в том смысле, как в модели BSP. Вместо этого итеративно планируется выполнение элемента графа, собираются состояния его соседей (их множество называется областью действия), вычисляется состояние элемента, после чего обновляются состояния соседей в области действия и список элементов графа, нуждающихся в планировании. Вычисление заканчивается, когда в графе не останется элементов, ожидающих планирования.

Комбинации моделей программирования и вычисления определяют пространство проектирования, состоящее из девяти вариантов. Но в настоящий момент существуют системы не для каждого варианта. Основные усилия исследователей и разработчиков направлены на BSP-системы, ориентированные на вершины (раздел 10.4.4), так что этот класс мы обсудим подробнее, чем другие. В тех случаях, когда соответствующая система пока не создана, мы кратко отметим, как она могла бы выглядеть.

---

[1] Мы употребляем термины глобальный барьер, глобальный барьер синхронизации и барьер синхронизации как синонимы.

10.4.4. Ориентированная на вершины пошагово-синхронная модель

Как уже было сказано, в ориентированных на вершины системах программист должен сфокусироваться на вычислениях в каждой вершине; ребра в таких системах не являются полноправными объектами, потому что для них никаких вычислений не выполняется. При следовании модели BSP вычисления в таких системах производятся итеративно (супершагами), так что на каждой итерации каждая вершина *v* читает состояние из отправленных ей на предыдущем супершаге сообщений, вычисляет на его основе свое новое состояние и передает его соседям (которые прочтут его на следующем супершаге). Затем система ждет, пока все машины-исполнители закончат вычисления на текущей итерации (глобальный барьер), после чего переходит к следующей итерации.

Каждая вершина «активна» или «неактивна». Вычисление начинается, когда все вершины активны, и останавливается, когда во всех них будет удовлетворено условие сходимости, все вершины перейдут в неактивное состояние и в системе не останется ожидающих обработки сообщений (рис. 10.25). Когда в некоторой вершине выполнено условие сходимости, она отправляет сообщение «голосую за остановку», а затем становится неактивной. Вершина остается неактивной, пока не получит извне сообщение снова активизироваться.

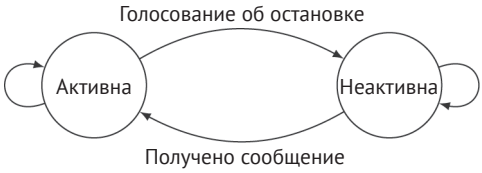


Рис. 10.25 ❖ Состояния вершин в системах, ориентированных на вершины

Мы уже говорили, что системы такого рода были очень популярны среди разработчиков. Классическими примерами являются Pregel и ее аналог с открытым исходным кодом Apache Giraph. Из других отметим GPS, Mizan, LFGraph, Pregelix и Trinity. При обсуждении деталей мы возьмем за образец систему Pregel (все такие системы называются «Pregel-подобными»).

Программист должен написать выполняемую в каждой вершине функцию `Compute()`, принимая во внимание семантику приложения. Система предоставляет встроенные функции `GetValue()` и `WriteValue()` для чтения и изменения состояния, ассоциированного с вершиной, а также функцию `SendMsg()`, которая отправляет обновленное состояние соседним вершинам. Располагая этой базовой функциональностью, программист может сосредоточиться на вычислении, выполняемом в каждой вершине. В некотором смысле этот подход напоминает MapReduce, где ожидается, что программист напишет код функций `map()` и `reduce()`, а сама система обеспечит механизм выполнения и взаимодействия.

Функция `Compute()` весьма общая; помимо вычисления нового состояния вершины, она может инициировать изменения структуры графа (так называемые *мутации*), если система это поддерживает. Например, алгоритм кластеризации может заменить множество вершин одной вершиной. Мутации, выполняемые на некотором супершаге, производятся в начале следующего супершага. Естественно, могут возникать конфликты, когда несколько вершин затребуют одну и ту же мутацию, например добавление вершины в одно и то же место графа, но с разными значениями. Эти конфликты разрешаются путем частичного упорядочения операций и реализации определенных пользователем обработчиков. Частичный порядок устанавливается следующим образом: сначала выполняется удаление ребер, затем удаление вершин, потом добавление вершин и, наконец, добавление ребер. Все мутации предшествуют вызову функции `Compute()`.

Пример 10.8. Для демонстрации ориентированной на вершины модели BSP вычислим компоненты связности графа, показанного на рис. 10.26а. Для этого примера возьмем граф попроще, чем в примере разбиения (рис. 10.22а), чтобы шаги вычисления были нагляднее.

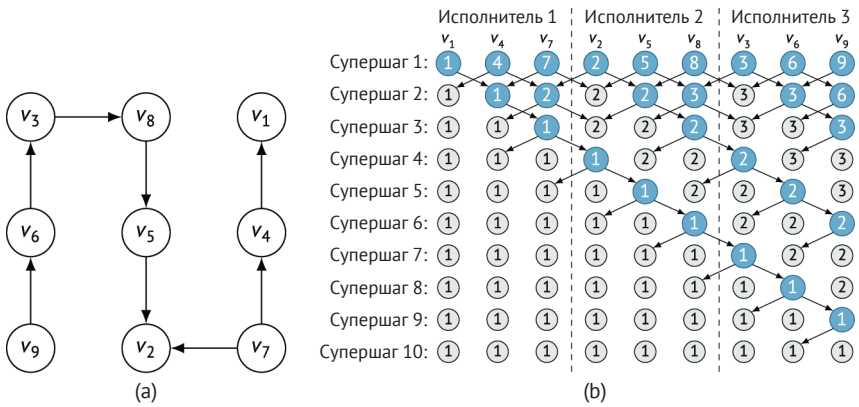


Рис. 10.26 ❖ Ориентированная на вершины модель BSP:
(a) пример графа; (b) вычисление КСС (серые вершины неактивны, синие активны)

Заметим, что поскольку этот граф ориентированный, под вычислением компоненты связности понимается вычисление компоненты слабой связности (КСС), когда ориентация игнорируется (см. пример 10.5). Кроме того, поскольку этот граф вполне связный, все вершины должны находиться в одной группе, и мы воспользуемся этим фактом для проверки правильности вычислений.

Ниже описан алгоритм вычисления КСС для ориентированной на вершины модели BSP. В каждой вершине хранится информация о группе, которой она принадлежит, и на каждом супершаге она разделяет эту информацию со своими соседями. В начале следующего супершага каждая вершина получает идентификаторы групп от соседей и выбирает наименьший из них как свой новый идентификатор группы, т. е. функция `Compute()` вычисляет

min{идентификаторы групп соседей, идентификатор своей группы}. Если идентификатор группы не изменился по сравнению с предыдущим супершагом, то вершина становится неактивной (т. е. в ней удовлетворено условие сходимости). В противном случае новый идентификатор группы рассылается соседям. Став неактивной, вершина перестает посылать сообщения соседям, но продолжает получать сообщения от активных соседей, чтобы узнать, не должна ли она снова активизироваться. Описанное вычисление продолжается на протяжении нескольких супершагов.

Ход вычисления показан на рис. 10.26b. На этапе инициализации алгоритм помещает каждую вершину в отдельную группу, используя в качестве идентификатора группы идентификатор самой вершины (например, вершина v_1 попадает в группу 1). Значением каждой вершины является состояние в конце супершага. Это состояние рассылается соседям. Стрелки показывают, когда сообщение потребляется узлом-получателям – поскольку стрелка направлена на следующий супершаг, сообщение не читается до его начала вне зависимости от того, когда оно отправлено или получено. Заметим, что на супершаге 1 вершины v_4, v_7, v_5, v_8, v_6 и v_9 изменяют свой идентификатор группы, тогда как в вершинах v_1, v_2 и v_3 значение не изменяется, поэтому они становятся неактивны.

В некоторых случаях вершина, бывшая неактивной, становится активной в результате полученных от соседей сообщений. Например, вершина v_2, которая стала неактивной на супершаге 2, становится активной на супершаге 4, когда получает от вершины v_7 сообщение с идентификатором группы 1, которое заставляет ее изменить идентификатор своей группы. Это отличительная особенность такого класса вычислений. ♦

Напомним, что эти системы выполняют параллельные вычисления в кластере, состоящем из мастер-узла и нескольких узлов-исполнителей, причем каждый исполнитель владеет множеством вершин графа и реализует функцию `Compute()`. В некоторых системах (например, GPS и Giraph) имеется дополнительная функция `Master.Compute()`, которая позволяет выполнять некоторые части алгоритма последовательно на мастер-узле. Эти функции придают дополнительную гибкость реализации алгоритма и открывают возможность для некоторых оптимизаций, которые мы обсудим ниже.

В некоторых алгоритмах важно запоминать глобальное состояние графа. Для этой цели можно реализовать *агрегатор*. Каждая вершина сообщает агрегатору свое значение, а результат агрегирования становится доступен всем вершинам на следующем супершаге. Обычно системы предоставляют несколько простых агрегаторов, например `min`, `max` и `sum`.

На производительность систем этой категории влияют два фактора: затраты на коммуникацию и число супершагов. Они, в свою очередь, зависят от свойств реальных графов, которые мы уже упоминали.

1. Степенные графы с несимметричным распределением вершин. Проблема заключается в том, что исполнители, владеющие вершинами большой степени, получают и должны обработать гораздо больше сообщений, чем прочие узлы, что ведет к несбалансированности нагрузки и появлению отстающих узлов.

2. Высокая средняя степень вершин. В результате каждая вершина должна обрабатывать много сообщений и взаимодействовать с большим количеством соседей, что влечет за собой высокие накладные расходы на коммуникацию.
3. Большой диаметр. Если в модели BSP каждый супершаг соответствует одному переходу (т. е. сообщению) между вершинами, то для завершения вычислений понадобится много супершагов, их число пропорционально диаметру графа. Хотя несколько сотен переходов не кажется такой уж большой величиной, не забывайте, что во многих аналитических задачах требуется выполнить несколько проходов по вершинам графа, что увеличивает вычислительную сложность алгоритма. Например, сообщалось, что для нахождения компонент сильной связности в графе диаметром 20 требуется свыше 4500 супершагов (без оптимизации).

Систему можно оптимизировать для уменьшения затрат на коммуникацию между узлами, воспользовавшись *комбинатором*, который объединяет сообщения, предназначенные вершине *v*, руководствуясь семантикой приложения (например, если *v* нужна только сумма значений соседей). Автоматически сделать это нельзя, потому что система не может определить, когда и как следует выполнять такое агрегирование. Поэтому предоставляется функция `Combine()`, которую должен написать программист.

Системная оптимизация для борьбы с асимметрией заключается в реализации алгоритмов разбиения графа, учитывающих асимметрию. Системы на основе разбиения, которые мы будем обсуждать в разделах 10.4.7–10.4.9, также решают эту проблему, но принципиально другим способом.

Есть предложения, позволяющие справиться и с этими проблемами, но в них требуется изменить реализацию алгоритмов обработки рабочей нагрузки. Не вдаваясь в детали, приведем один пример для иллюстрации. В некоторых алгоритмах анализа графов небольшая часть вершин может оставаться активной, после того как все остальные перешли в неактивное состояние, и в результате число супершагов, необходимых для сходимости, сильно возрастает. Оптимизация предлагает, в случае когда доля «активных» вершин достаточно мала, перенести вычисление на мастер-узел и выполнить его последовательно с помощью функции `Master.Compute()`. Экспериментально показано, что число супершагов при этом уменьшается на 20–60 %.

10.4.5. Ориентированная на вершины асинхронная модель

В этих системах модель программирования такая же, как в предыдущем случае, но модель синхронных вычислений ослаблена, хотя барьеры синхронизации в конце каждого супершага сохраняются. Следовательно, для каждой вершины на каждом супершаге, как и выше, вычисляется функция `Compute()`, и результаты проталкиваются соседям, но этой функции доступны сообщения, отправленные не только на предыдущем, но и на текущем супершаге.

Сообщения, недоступные в момент выполнения `Compute()`, подбираются в начале следующего супершага, как в модели BSP. Этот подход решает важную проблему модели BSP, не жертвуя простотой ориентированного на вершины программирования: вершина может видеть недавние сообщения сразу, не дожидаясь следующего супершага. Обычно это приводит к более быстрой сходимости, чем в системах на основе BSP. Примерами могут служить системы GRACE и GiraphUC.

Пример 10.9. Для демонстрации ориентированной на вершины модели AP воспользуемся примером вычисления компонент слабой связности из предыдущего примера 10.8. Для простоты предположим, что все сообщения, которыми обмениваются вершины, достигают места назначения на том же супершаге, на котором отправлены, и что вычисление в каждом узле-исполнителе также завершается на том же супершаге. Кроме того, предположим, что все исполнители работают в однопоточном режиме, т. е. в каждый момент времени `Compute()` вычисляется только для одной вершины. При таких предположениях вычисление КСС для графа на рис. 10.26а производится, как показано на рис. 10.27. Заметим, к примеру, что на этапе инициализации v_1 отправляет свой идентификатор группы (1) вершине v_4, а v_4 отправляет свой идентификатор группы (4) вершинам v_1 и v_7. Поскольку вычисление, по предположению, однопоточное, v_4.`Compute` изменяет идентификатор группы v_4 на 1, и на том же супершаге (1) новый идентификатор отправляется v_7. Поэтому в момент вычисления v_7.`Compute()` идентификатор группы v_7 уже установлен в 1. Доступ к новому состоянию соседних вершин на том же супершаге – отличительная особенность модели AP. ♦

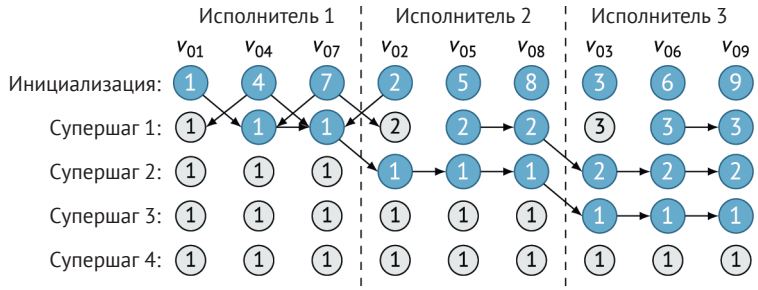


Рис. 10.27 ❖ Ориентированная на вершины модель AP

Как уже было отмечено, при асинхронном выполнении необходим контроль согласованности с помощью распределенных блокировок. В системах рассматриваемого класса это выглядит следующим образом. К состоянию некоторой вершины могут обращаться другие вершины во время выполнения своей функции `Compute()` – вершина v_i может быть занята выполнением функции `Compute()`, когда соседняя вершина v_j отправляет ей свои обновления. Чтобы избежать этого, на каждую вершину ставятся блокировки. Поскольку вершины распределены по узлам-исполнителям, для обеспечения согласованности необходим распределенный механизм блокировки.

Еще одна проблема модели AP заключается в нарушении другой гарантии согласованного выполнения, предоставляемой BSP. Каждая вершина обрабатывает в `Compute()` все сообщения, полученные с момента последнего выполнения, и среди них будут как старые сообщения (с предыдущего супершага), так и новые (с текущего супершага). Поэтому нельзя утверждать, что на каждом супершаге каждая вершина согласованно вычисляет новое состояние, основываясь на состояниях соседних вершин в конце предыдущего супершага. Например, такое ослабление позволяет вершине выполнять свою функцию `Compute()`, как только она получит высокоприоритетное сообщение.

Хотя модель AP решает проблему задержек в BSP, ей все же свойственны узкие места из-за наличия глобальных барьеров синхронизации, а именно высокие затраты на коммуникацию и необходимость мириться с отстающими узлами. Эти проблемы можно преодолеть, устранив часть барьеров, как предложено в *безбарьерной асинхронной параллельной* (barrierless asynchronous parallel – BAP) модели, принятой в GiraphUC. В BAP сохраняются *глобальные барьеры* между *глобальными супершагами*, когда узлы-исполнители глобально синхронизируются, но каждый глобальный супершаг разбивается на *логические супершаги*, разделенные *локальными барьерами* с очень низкими накладными расходами. Это позволяет быстрым исполнителям вычислить функцию `Compute()` несколько раз (по одному на каждом логическом супершаге), прежде чем возникнет необходимость глобальной синхронизации с более медленными исполнителями. Как и в модели AP, вершины могут сразу читать полученные локальные и удаленные сообщения, что уменьшает задержку обработки сообщений. На рис. 10.28 показана модель BAP с тремя исполнителями; первый (W_1) выполняет четыре логических супершага (ЛСШ) в одном глобальном супершаге (ГСШ), второй – два, а третий – три. Пунктирными стрелками показаны сообщения, полученные и обработанные на одном логическом супершаге, а сплошными – сообщения, подобранные на следующем глобальном супершаге.

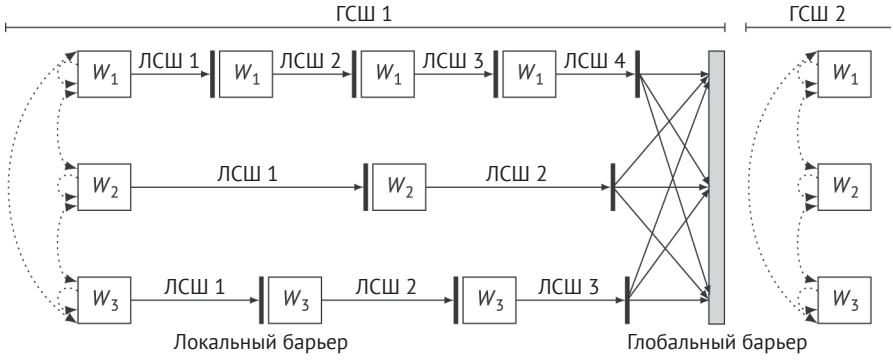


Рис. 10.28 ❖ Модель BAP с тремя узлами-исполнителями

В модели BAP нужно аккуратно определять условие завершения. Напомним, что в моделях BSP и AP условие завершения проверяется у каждого барьера синхронизации – все вершины должны быть неактивны, и в системе

не должно остаться необработанных сообщений. Но теперь, когда появились локальные и глобальные барьеры, эту проверку нужно делать в обоих случаях. Можно поступить просто – у локального барьера проверять, есть ли еще необработанные локальные или глобальные сообщения; если нет, то больше делать нечего, и вершины в этом узле-исполнителе подошли к глобальному барьеру. Когда все вершины графа окажутся у глобального барьера, производится вторая проверка – такая же, как в BSP и AP: вычисление завершается глобально, если все вершины неактивны и не осталось сообщений.

10.4.6. Ориентированная на вершины модель сбора–обработки–распространения

К системам этого типа относится GraphLab. Как уже отмечалось, у такого подхода есть синхронная версия (реализованная в GraphLab Sync), практически не отличающаяся от ориентированной на вершины модели BSP (если не считать нюанса, связанного с вытягиванием), поэтому его мы обсуждать не будем. Асинхронная версия устроена по-другому: на этапе сбора вершина вытягивает данные у своих соседей, а не соседи проталкивают данные ей[1]. Для каждой вершины v определена *область действия* [$Scope(v)$], которая состоит из данных, хранящихся во всех смежных ребрах и вершинах, а также данных в самой v. Функция `Compute()` (в GraphLab она называется `Update`) принимает v и $Scope(v)$ и возвращает обновленную область $Scope'(v)$, а также множество вершин V' с изменившимся состоянием, которые являются кандидатами на планирование. Алгоритм, принимающий граф G и начальное множество вершин V', состоит из трех шагов:

1) удалить вершину из V' в соответствии с решением планировщика;
2) выполнить функцию `Compute()` и вычислить $Scope'(v)$ и V';
3) $V' \leftarrow V \cup V'$.

Эти три шага выполняются итеративно, пока в V' еще остаются вершины. Отделение обновления состояний в $Scope'(v)$ (т. е. состояний соседей) от планирования вычислений в вершинах – это основное отличие модели GAS от AP, в которой сообщения об обновлении состояний вершин служат также для планирования вычислений в этих вершинах. Такое разделение позволяет гибко выбирать порядок обсчета вершин, например на основе приоритетов или с учетом балансировки нагрузки. Заметим также, что в GAS нет явной функции `SendMsg()`; разделение измененных состояний производится на этапе сбора.

Поскольку вершина v может напрямую читать данные из своей области $Scope(v)$, возможно возникновение несогласованности, т. к. в процессе вычислений в нескольких вершинах могут быть выполнены конфликтующие обновления состояния. Для решения этой проблемы, как уже было сказано, требуется развернуть механизмы распределенной блокировки. Когда вершина v выполняет `Compute`, она ставит блокировку на свою область $Scope(v)$,

[1] GraphLab отличается также реализацией распределенной разделяемой памяти, но сейчас нам это не важно.

производит вычисление, обновляет *Scope*(*v*) и затем снимает блокировку. В GraphLab это называется полной согласованностью. В этой конкретной системе есть еще два уровня ослабленной согласованности для приложений, семантика которых не требует полного взаимного исключения: *реберная согласованность* и *вершинная согласованность*. Реберная согласованность гарантирует, что *v* имеет доступ для чтения-записи к собственным данным и данным смежных с ней ребер, но только доступ для чтения к соседним вершинам. Например, для вычисления PageRank достаточно реберной согласованности, поскольку нужно только читать ранги соседних вершин. Вершинная согласованность гарантирует, что пока *v* выполняет свою функцию `Compute()`, никакая другая вершина не сможет к ней обратиться.

10.4.7. Ориентированная на разделы пошагово-синхронная модель

В разделе 10.4.4 мы уже отмечали, что многие реальные графы обладают свойствами, которые затрудняют применение ориентированных на вершины систем, поэтому были разработаны оптимизации для решения возникающих проблем. BSP-системы, ориентированные на разделы, предлагают совершенно другой подход к этим проблемам. В них используется такое разбиение графа по узлам-исполнителям, что вместо взаимодействия каждой вершины с остальными с помощью передачи сообщений, как в ориентированной на вершины модели, сообщениями обмениваются блоки (разделы), применяя более простой последовательный алгоритм, реализованный в каждом разделе. Вычисление устроено, как в модели BSP, т. е. выполняются супершаги, до тех пор пока алгоритм не сойдется. Этот подход принят в системах Blogel и Giraph++.

Важнейшей особенностью этих систем является то, что они выполняют последовательный алгоритм внутри блоков, и взаимодействие происходит только на уровне блоков. Можно взглянуть на это следующим образом: пусть дан граф $G = (V, E)$, тогда после разбиения мы имеем граф $G' = (B, E')$, где B – множество блоков, а E' – множество ребер между блоками. В алгоритмах, ориентированных на разделы, затраты на коммуникацию ограничены сверху величиной $|E'|$, существенно меньшей, чем $|E|$; поэтому для графов с высокой плотностью ребер затраты на коммуникацию ниже. Например, в эксперименте по вычислению компонент связности в графе социальной сети Friendster оказалось, что ориентированной на вершины системе понадобилось в 372 раза больше сообщений и в 48 раз больше времени, чем системе, ориентированной на разделы. К тому же в системах, ориентированных на разделы, уменьшается диаметр графа, потому что каждый блок представлен одной вершиной в G', и это ведет к значительному уменьшению числа супершагов в модели вычислений BSP. Аналогичный эксперимент по вычислению компонент связности графа дорожной сети США (диаметром приблизительно 9000) показал, что число супершагов уменьшилось с 6000 до 22. Наконец, проблема асимметричного распределения степеней вершин решается алгоритмом разбиения графа, который гарантирует, что число вершин в каждом блоке сбалансировано. Поскольку в каждом блоке выполняется последовательный

алгоритм, наличие вершин большой степени необязательно влечет за собой увеличение количества сообщений – опять же потому, что ориентированные на вершины системы работают с графом G, а ориентированные на разделы – с гораздо меньшим графом G'.

Пример 10.10. Рассмотрим, как вычисление КСС можно было бы выполнить в ориентированной на разделы BSP-системе. Шаги вычисления показаны на рис. 10.29, где части графа в каждом узле-исполнителе обозначены разными цветами. В каждом разделе выполняется последовательный алгоритм, поэтому необязательно использовать тот алгоритм, который мы обсуждали ранее, – когда в начале работы каждой вершине сопоставляется ее собственная группа, – но для сравнения с предыдущими подходами мы возьмем именно его. На супершаге 1 каждый узел-исполнитель выполняет последовательное вычисление с целью определить идентификаторы групп для вершин в своем разделе – для исполнителя 1 наименьший идентификатор группы равен 1, он и становится идентификатором группы для вершин v_1, v_4 и v_7. Аналогичное вычисление производится в других узлах. В конце супершага каждый исполнитель отправляет свой идентификатор группы другим исполнителям, и вычисление повторяется.

Рис. 10.29 ❖ Ориентированная на разделы модель BSP

Заметим, что число супершагов в этом случае такое же, как в ориентированной на вершины модели AP (пример 10.9); в общем случае оно могло бы быть меньше, но не это главное. Настоящая экономия достигается за счет снижения числа сообщений между узлами: в ориентированной на разделы системе передается всего 6 сообщений, тогда как в ориентированной на вершины – 20 сообщений. В графе, рассматриваемом в этих примерах, плотность связей невелика, в противном случае экономия была бы гораздо больше. ♦

10.4.8. Ориентированная на разделы асинхронная модель

Системы такого вида должны были бы разбивать граф между узлами-исполнителями, как описано в разделе 10.4.7, и выполнять последовательный

алгоритм в каждом разделе, но взаимодействие между исполнителями при передаче сообщений асинхронное. Мы уже отмечали, что асинхронное взаимодействие обычно реализуется с помощью распределенной блокировки. В этом смысле такие системы были бы очень похожи на распределенные СУБД, обсуждаемые в этой книге, если рассматривать каждый раздел как фрагмент данных. В настоящее время не реализовано ни одной системы из этой категории.

10.4.9. Ориентированная на разделы модель сбора–обработки–распространения

Единственное отличие этого случая от ориентированной на разделы модели BSP (см. раздел 10.4.7) – использование модели GAS, основанной на вытягивании, а не модели BSP на основе проталкивания. И снова отметим, что системы этого класса очень похожи на распределенные СУБД, нужно только внести необходимые изменения в операции передачи данных в планах выполнения запросов. До сих пор ни одной системы, принадлежащей этой категории, не разработано.

10.4.10. Ориентированная на ребра пошагово-синхронная модель

В системах, ориентированных на ребра, главным объектом является ребро, и в этом смысле они двойственны системам, ориентированным на вершины. Вычисления производятся в каждой вершине, а супершаги повторяются до достижения сходимости. Заметим, однако, что ребро графа идентифицируется двумя инцидентными ему вершинами. Поэтому выполнение функции `Compute()` для ребра требует выполнения в вершинах, инцидентных этому ребру. Таким образом, настоящее отличие заключается в том, что система обрабатывает за раз по одному ребру, а не по одной вершине, как в системах, ориентированных на вершины.

Возникает естественный вопрос: в чем здесь выгода, если в большинстве графов ребер гораздо больше, чем вершин? На первый взгляд кажется, что в системах, ориентированных на ребра, вычислений должно быть гораздо больше. Вспомним, однако, в ориентированных на вершины системах чем больше ребер, тем выше стоимость передачи сообщений. Кроме того, ребра в таких системах обычно отсортированы по начальным вершинам, по которым для ускорения доступа строится индекс. Когда обновления состояния распространяются в соседние вершины, по этому индексу производится произвольный доступ для поиска ребер, инцидентных вершинам, и этот доступ обходится дорого. В системах, ориентированных на ребра, эти проблемы решаются тем, что обработке подвергается неотсортированная последовательность ребер, в которой каждое ребро определяет свою начальную и конечную вершины, – нет никакого произвольного доступа по индексу и, когда по за-

вершении вычисления в ребре рассылаются сообщения, существует только одна конечная вершина. Поскольку мы рассматриваем модель вычислений BSP, обновления, вычисленные на супершаге, становятся доступны в начале следующего супершага. Этот подход реализован в параллельной системе с общей памятью X-Stream, где имеются также оптимизации для обработки графов в памяти и на диске.

10.4.11. Ориентированная на ребра асинхронная модель

Системы этого класса были бы модификациями ориентированных на вершины асинхронных систем (раздел 10.4.5), только требовалось бы обеспечить согласованное выполнение в ребрах, а не в вершинах. Поэтому блокировки ставились бы на ребра, а не на вершины. В настоящее время не существует систем, принадлежащих этой категории.

10.4.12. Ориентированная на ребра модель сбора–обработки–распространения

Комбинация ориентированной на ребра модели программирования с моделью вычислений GAS повлекла бы за собой такие же изменения в ориентированных на ребра BSP-системах (раздел 10.4.10), как при переходе от ориентированных на вершины BSP-систем к ориентированным на вершины GAS-системам. Это означало бы, что функции сбора, обработки и распространения нужно было бы реализовать на ребрах, так что основанное на вытягивании чтение состояний производится на этапе сбора. На данный момент неизвестно ни одной системы такого рода.

10.5. Озера данных

Технологии больших данных позволяют хранить и анализировать данные самых разных видов: структурированные, неструктурированные или слабо структурированные в их естественном формате. Это дает возможность по-новому решить старую задачу о физической интеграции данных (см. главу 7), которая обычно решалась с помощью хранилищ данных. В хранилище данные из различных источников приводятся к единому формату, обычно реляционному, для чего данные необходимо преобразовывать. Напротив, в озере данные хранятся в своем «родном» формате с использованием хранилища больших данных, например HDFS. Каждый элемент данных можно хранить непосредственно, сопроводив уникальным идентификатором и метаданными (например, источник данных, формат и т. д.). Поэтому необходимости в преобразовании данных не возникает. Технология обещает, что для

каждого интересующего предприятие вопроса можно будет быстро найти релевантный набор данных и проанализировать его.

Термин «озеро данных» введен для противопоставления хранилищам данных и витринам данных. Часто он ассоциируется с программной экосистемой Hadoop. Тема стала очень актуальной, но, если сравнивать с хранилищами данных, вызывает много споров. Далее в этом разделе мы сопоставим озера и хранилища данных, затем опишем принципы и архитектуру озер данных и, наконец, расскажем об открытых вопросах.

10.5.1. Озера данных и хранилища данных

В главе 7 мы говорили, что хранилища данных призваны обеспечить физическую интеграцию баз данных. Это основа OLAP-приложений и бизнес-аналитики, на них построена ориентированная на данные стратегия предприятия. Когда хранилища данных только появились (в 1980-х годах), данные предприятия хранились в оперативных OLTP-базах. Сегодня все больше и больше полезных данных поступает из разнообразных источников больших данных: журналов веб-серверов, социальных сетей и электронной почты. В результате выявились недостатки традиционных хранилищ данных.

- **Длительный процесс разработки.** Разработка хранилища данных может длиться годами. Основная причина в том, что требуется заранее точно определить и смоделировать необходимые данные. После того как необходимые данные найдены, часто на «свалке» разрозненных данных предприятия, следует аккуратно определить глобальную схему и ассоциированные метаданные, а затем спроектировать процедуры очистки и трансформации данных.
- **Определение схемы при записи.** Обычно хранилище данных опирается на реляционную СУБД, и его структура описывается реляционной схемой. В реляционных СУБД принят подход, который недавно стали называть «schema-on-write» в противоположность «schema-on-read» (см. ниже). В этом случае данные записываются в базу в фиксированном формате, определенном схемой. Это помогает добиться согласованности. Затем пользователи в соответствии со схемой выражают запросы на выборку данных, которые уже представлены в правильном формате. Обработка запроса будет эффективной, поскольку разбирать данные во время выполнения нет необходимости. Однако при этом становится более сложной и дорогой адаптация бизнес-среды к изменяющимся условиям. Например, при появлении новых данных часто требуется модифицировать схему (скажем, добавлять новые столбцы), что, в свою очередь, может повлечь за собой модификацию приложений и предопределенных запросов.
- **Обработка OLAP-задач.** Хранилище данных обычно оптимизировано для рабочих нагрузок типа OLAP, когда аналитики интерактивно опрашивают данные по различным измерениям, например с помощью кубов данных. В большинстве OLAP-приложений, например анализе тенденций и прогнозировании, нужно анализировать исторические

данные, а их актуальные версии не представляют интереса. Но недавно появились OLAP-приложения, которым нужен доступ к оперативным данным в реальном масштабе времени, что довольно трудно поддержать в хранилищах данных.

- **Трудоемкая разработка с применением ETL.** Для интеграции гетерогенных источников данных в глобальную схему необходимы сложные ETL-программы для очистки, преобразования и обновления данных. По мере диверсификации источников разрабатывать такие программы становится все труднее.

Озеро данных представляет собой центральный репозиторий всех данных предприятия в их естественном формате. Как и хранилище данных, его можно использовать для OLAP-приложений и бизнес-аналитики, но также для пакетного или оперативного анализа данных с применением технологий больших данных. По сравнению с хранилищами данных озеро предлагает следующие преимущества.

- **Определение схемы при чтении.** Термином «schema-on-read» обозначают подход к анализу больших данных, когда во главу узла ставятся загруженные данные, как в Hadoop. В этом случае данные загружаются как есть, в своем «родном» формате, например в файловую систему Hadoop HDFS. А уже потом, во время чтения данных, на них накладывается схема для выделения представляющих интерес полей. Таким образом, данные можно опрашивать в естественном формате. Это резко повышает гибкость, поскольку в озеро в любой момент можно добавить новые данные. Однако требуется больше усилий для написания кода, применяющего к данным схему; например, его можно включить в функцию Map каркаса MapReduce. Разбор данных также приходится проводить во время выполнения запроса.
- **Обработка различных рабочих нагрузок.** Программный стек управления большими данными (см. рис. 10.1) поддерживает разные методы доступа к одним и тем же данным, например: пакетный анализ с помощью каркаса типа MapReduce, интерактивные OLAP-приложения или бизнес-аналитика с помощью каркаса типа Spark, анализ в реальном времени с помощью каркасов потоковой обработки данных. Агрегируя различные каркасы, озеро данных может поддержать обработку рабочих нагрузок разных типов.
- **Экономически эффективная архитектура.** Опираясь на кластеры без разделения ресурсов и на программы с открытым исходным кодом для реализации программного стека управления большими данными, озеро данных обеспечивает отличные показатели соотношения стоимости и производительности и отдачи на капитал.

10.5.2. Архитектура

Озеро данных должно предоставлять следующие основные возможности:

- собирать все полезные данные: исходные, преобразованные, поступающие из внешних источников и т. д.;

- позволять пользователям из различных подразделений предприятия исследовать данные и обогащать их метаданными;
- использовать для доступа к данным различные методы: пакетные, интерактивные, в режиме реального времени и т. д.;
- осуществлять руководство данными[1], обеспечивать безопасность, управлять данными и задачами.

На рис. 10.30 показана архитектура и основные компоненты озера данных. В центре мы видим компоненты управления большими данными (хранение данных, доступ к данным, анализ данных и управление ресурсами), поверх них могут быть построены различные презентации и приложения. Эти компоненты входят в состав программного стека управления большими данными, их можно найти среди программ с открытым исходным кодом, спонсируемых фондом Apache. Заметим, что в настоящее время имеется много инструментов бизнес-аналитики (БА), работающих с Hadoop, и среди них есть как совсем новые, так и расширения традиционных инструментов БА для РСУБД. Можно также выделить два подхода (которые могут быть объединены в одном инструменте):

1) SQL поверх Hadoop, т. е. использование драйвера Hadoop SQL, например HiveQL или Spark SQL. Примерами могут служить Tableau, Platfora, Pentaho, Power BI и DB2 BigSQL;
2) библиотека функций для доступа к HDFS с помощью высокоуровневых операторов, например Datameer, Power BI и DB2 BigSQL.

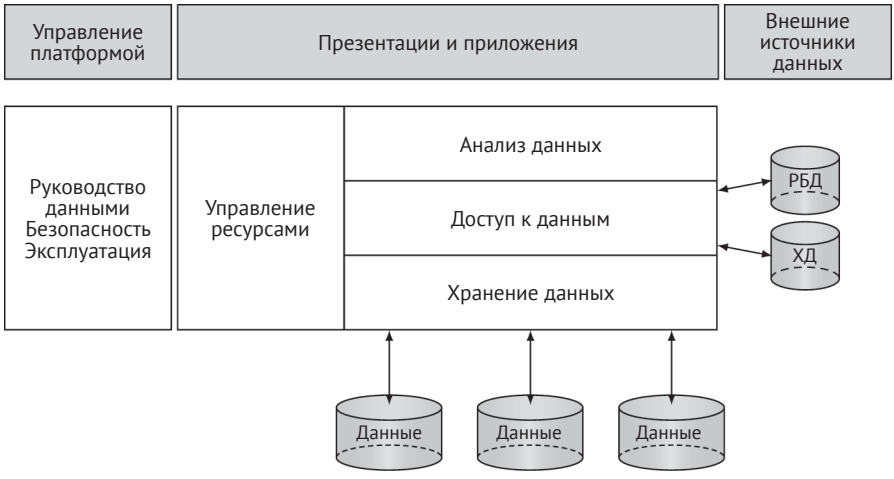


Рис. 10.30 ❖ Архитектура озера данных

В левой части рисунка расположены средства управления платформой: руководство данными, безопасность данных и эксплуатационные средства. Эти компоненты дополняют управление большими данными функциями,

[1] Смысл термина «руководство данными» описан в статье Википедии по адресу https://ru.wikipedia.org/wiki/Data_Governance. – *Прим. перев.*

необходимыми для разделения данных в масштабе предприятия (между различными подразделениями). Руководство данными играет все большую роль в озерах данных, потому что оно необходимо для управления данными в соответствии с политикой предприятия – с особым вниманием к законам о конфиденциальности данных, например знаменитому Генеральному регламенту по защите персональных данных, принятому Европейским союзом в мае 2018 года. Обычно эту политику определяет комитет по руководству данными, а реализуют распорядители данных, отвечающие за организацию данных для нужд предприятия. К безопасности данных относятся аутентификация пользователей, контроль доступа и защита данных. В состав эксплуатационных средств входят подготовка, мониторинг и планирование задач (обычно в кластере без разделения ресурсов). Среди программных продуктов Apache можно найти также инструменты для руководства данными, например Falcon, обеспечения безопасности данных, например Ranger и Sentry, и эксплуатации, например Ambari и Zookeeper.

Наконец, справа показано, что различные виды внешних источников, например SQL, NoSQL и т. д., можно интегрировать, обычно с помощью средств для доступа к данным, например соединителей Spark.

10.5.3. Проблемы

Построение и эксплуатация озера данных остаются трудной задачей как по методологическим, так и по техническим причинам. Методология организации хранилища данных сейчас изучена и понятна. Она включает сочетание регламентированного моделирования данных (определение схемы при записи), управление метаданными и руководство данными – все вместе это обеспечивает сильную согласованность данных. Затем, применяя мощные инструменты OLAP или бизнес-аналитики, различные пользователи, даже не обладающие глубокими познаниями в области анализа данных, могут извлекать из данных пользу. В частности, витрина данных упрощает анализ данных, относящихся к конкретной потребности организации.

С другой стороны, в озере данных нет никакой согласованности, что значительно затрудняет анализ данных в масштабе предприятия. Это основная причина, по которой нужны специалисты по обработке данных и распорядители данных. Еще одна причина заключается в том, что технологический ландшафт больших данных сложен и продолжает изменяться. Поэтому при построении озера данных рекомендуется придерживаться следующей методологии, подтвержденной практикой:

- составить список приоритетов и преимуществ по сравнению с корпоративным хранилищем данных. Он должен включать точное определение бизнес-целей и соответствующие требования к данным в озере;
- выстроить глобальное видение архитектуры озера данных, которая должна быть расширяемой (чтобы адаптироваться к эволюции технологий), а также предусматривать руководство данными и управление метаданными;

- определить политики безопасности и конфиденциальности, что особенно важно, если данные разделяются между различными направлениями бизнеса;
- определить модель вычислений и хранения, поддерживающую глобальное видение. В частности, расширяемость и масштабируемость должны быть определяющими аспектами при выборе технических решений;
- определить производственный план, включающий соглашения об уровне обслуживания (service level agreements – SLA) в терминах времени непрерывного функционирования, объема, разнородности, скорости возникновения и достоверности.

Технические причины затруднений при создании озер данных связаны с интеграцией и качеством данных. При традиционной интеграции данных (см. главу 7) акцент делается на проблеме интеграции схем, в т. ч. на сопоставлении и отображении схем, с целью создать глобальную схему. В контексте интеграции больших данных, когда имеется много гетерогенных источников, проблема интеграции схем осложняется. Озеро данных попросту уходит от проблемы интеграции схем, осуществляя управление бессхемными данными. Однако со временем вопрос об интеграции все же может возникнуть, если потребуется повысить степень согласованности данных. В таком случае интерес может представить автоматическое извлечение метаданных и информации о схеме из многих взаимосвязанных элементов данных, например составляющих один набор данных. Один из подходов к этой задаче – комбинирование методов машинного обучения, сопоставления и кластеризации.

10.6. Заключение

Большие данные и их роль в науке о данных стали важными темами в управлении данными, хотя точно определить, что это такое, затруднительно. Не существует унифицированной инфраструктуры, в рамках которой можно было представить разработки в этой области; быть может, эталонная архитектура на рис. 10.1 – лучшее, что можно сделать. Поэтому в этой главе мы сосредоточились на свойствах, характеризующих эти системы, и на основных платформах, предложенных для работы с ними. Мы обсудили распределенные системы хранения и платформы обработки MapReduce и Spark, которые решают проблемы управления большими объемами данных и их обработки. Мы рассмотрели потоки данных, позволяющие справиться с высоким темпом поступления данных. Мы рассказали о средствах анализа графов, которые наряду с потоками данных решают проблему разнородности. При обсуждении озер данных мы остановились на вопросах разнородности и масштабируемости в свете интеграции данных; попутно высветились проблемы качества данных (достоверность) и необходимость в очистке, если исходные данные плохо подготовлены изначально.

Темы, рассмотренные в этой главе, неизбежно продолжат эволюционировать – это та область, в которой технологии развиваются особенно быстро.

Мы заложили фундамент и привели ссылки на основополагающие работы. В следующем разделе мы дадим дополнительные ссылки, но настоятельно рекомендуем читателю самостоятельно следить за свежими публикациями.

10.7. Библиографические замечания

Статистические сведения по большим данным разрознены, не существует одной публикации, содержащей полную статистику. Статистика YouTube взята из работы [Brewer et al. 2016], а статистика Alibaba – из личной переписки. В блогах и статьях в вебе можно найти гораздо больше информации.

Из ранних обсуждений проблем поддержки СУБД со стороны операционной системы отметим работу [Stonebraker 1981], в которой также объясняется, почему первые СУБД перешли от хранения на основе файловой системы к блочному хранению. Что касается более современных систем хранения, которые мы обсуждали, то Google File System описана в работе [Ghemawat et al. 2003], а Ceph – в работе [Weil et al. 2006].

Наше обсуждение платформ обработки больших данных (в частности, MapReduce) основано главным образом на работе [Li et al. 2014]. В работах [Sakr et al. 2013] и [Lee et al. 2012] также имеются обзоры на эту тему. Оригинальное описание MapReduce приведено в работах [Dean and Ghemawat 2004, 2010]. Критика MapReduce обсуждается в работах [DeWitt et al. 2008, Dewitt and Stonebraker 2009, Pavlo et al. 2009, Stonebraker et al. 2010]. По языкам, поддерживающим MapReduce, имеются следующие источники: HiveQL [Thusoo et al. 2009], Tenzing [Chattopadhyay et al. 2011], JAQL [Beyer et al. 2009], Pig Latin [Olston et al. 2008], Sawzall [Pike et al. 2005]), FlumeJava [Chambers et al. 2010], SystemML [Ghoting et al. 2011]. Реализация алгоритма 1-Bucket-Theta соединения в MapReduce описана в работе [Okcan and Riedewald 2011], широковещательное соединение – в работе [Blanas et al. 2010], а соединение с повторным разбиением – в работе [Blanas et al. 2010].

Система Spark предложена в работах [Zaharia et al. 2010, Zaharia 2016]. Язык Spark SQL как часть экосистемы Spark обсуждается в работе [Armbrust et al. 2015], Spark Streaming – в работе [Zaharia et al. 2013], а GraphX – в работе [Gonzalez et al. 2014].

По системам потоков данных имеется обширная литература, в т. ч. несколько книг. В работе [Golab and Özsu 2010] рассматриваются преимущественно ранние системы, а также проблемы запросов и моделирования данных. В работе [Aggarwal 2007] обсуждается широкий круг тем (включая добычу данных в потоках, о которой мы не говорили) с акцентом на ранние работы. Дополнительное обсуждение добычи данных в потоках можно найти в работе [Bifet et al. 2018]. В работе [Muthukrishnan 2005] рассматриваются теоретические основания таких систем. Обобщение систем потоков данных на обработку событий – еще одно направление исследований; мы не обсуждали эту тему, но хорошей отправной точкой может служить работа [Etzion and Niblett 2010]. Помимо рассмотренных нами систем, имеются СПД, развернутые в облаке, примером может служить StreamCloud [Gulisano et al. 2010, 2012].

Определение потока данных как последовательности снабженных временными метками элементов, поступающих в некотором порядке, которая допускает только дописывание в конец, дано в работе [Guha and McGregor 2006]. Другие определения потока данных см. в работах [Wu et al. 2006, Tucker et al. 2003]. Идея исправленных кортежей, вставляемых в поток данных, сформулирована в работе [Ryvkina et al. 2006]. Семантика потоковых запросов обсуждается в работе [Arasu et al. 2006] в контексте языка CQL и в более общем виде – в работе [Law et al. 2004]. Эти языки классифицируются как декларативные (QL [Arasu et al. 2006, Arasu and Widom 2004], GSQL [Cranor et al. 2003], StreaQuel [Chandrasekaran et al. 2003]) или процедурные (Aurora [Abadi et al. 2003]). Выполнение операторов в потоковых системах – важный вопрос в связи с требованиями к отсутствию блокировок. Неблокирующее соединение – тема работ [Haas and Hellerstein 1999a, Urhan and Franklin 2000, Viglas et al. 2003, Wilschut and Apers 1991], агрегирование – работ [Hellerstein et al. 1997, Wang et al. 2003]. Соединение более двух потоков (многопотоковое соединение) обсуждается в работах [Golab and Özsu 2003, Viglas et al. 2003], а соединение потоков со статическими данными – в работе [Balazinska et al. 2007]. Вопрос о прерывателях как средстве разблокировки впервые рассмотрен в работе [Tucker et al. 2003]. Прерыватели также используются для уменьшения объема состояния, поддерживаемого операторами [Ding and Rundensteiner 2004, Ding et al. 2004, Fernández-Moctezuma et al. 2009, Li et al. 2006, 2005]. Пульсы как прерыватели, которые контролируют временные метки будущих кортежей, обсуждаются в работах [Johnson et al. 2005, Srivastava and Widom 2004a].

Обработка запросов к потокам данных – тема работ [Abadi et al. 2003, Adamic and Huberman 2000, Arasu et al. 2006, Madden and Franklin 2002, Madden et al. 2002a]. Обработка оконных запросов обсуждается в работах [Golab and Özsu 2003, Hammad et al. 2003a, 2005, Kang et al. 2003, Wang et al. 2004, Arasu et al. 2006, Hammad et al. 2003b, 2004]. Подходы к управлению нагрузкой, когда темп потока превышает возможности обработки, представлены в работах [Tatbul et al. 2003, Srivastava and Widom 2004b, Ayad and Naughton 2004, Liu et al. 2006, Reiss and Hellerstein 2005, Babcock et al. 2002, Cammert et al. 2006, Wu et al. 2005].

Несколько систем обработки потоков было предложено и разработано в качестве прототипов и производственных систем. Мы выделили системы управления потоками данных (СУПД): STREAM [Arasu et al. 2006], Gigascope [Cranor et al. 2003], TelegraphCQ [Chandrasekaran et al. 2003], COUGAR [Bonnet et al. 2001], Tribeca [Sullivan and Heybey 1998], Aurora [Abadi et al. 2003], Borealis [Abadi et al. 2005], а также системы обработки потоков данных (СОПД): Apache Storm [Toshniwal et al. 2014], Heron [Kulkarni et al. 2015], Spark Streaming [Zaharia et al. 2013], Flink [Carbone et al. 2015], MillWheel [Akidau et al. 2013] и TimeStream [Qian et al. 2013]. Как было отмечено, все СУПД, кроме Borealis, работают на одной машине, а все СОПД распределенные или параллельные.

Секционирование потоковых данных в параллельных и распределенных системах обсуждается в работах [Xing et al. 2006] и [Johnson et al. 2008]. Расщепление ключа предложено в работе [Azar et al. 1999], частичная группировка ключей (PKG) – в работе [Nasir et al. 2015] (принцип «выбора двух», на

котором основана PKG, обсуждается в работе [Mitzenmacher 2001]). Метод PKG был обобщен на выбор большего числа вариантов для головы распределения в работе [Nasir et al. 2016]. Применение гибридного секционирования для работы с асимметричными данными описано в работах [Gedik 2014, Pacaci and Özsu 2018]. Повторное секционирование между операциями плана запроса обсуждается в работах [Zhu et al. 2004, Elseidy et al. 2014, Heinze et al. 2015, Fernandez et al. 2013, Heinze et al. 2014]. Важнейший в этом контексте оператор Flux предложен в работе [Shah et al. 2003].

Семантика восстановления параллельных и распределенных потоковых систем – тема работы [Hwang et al. 2005].

Существует немало книг, посвященных конкретным аспектам анализа графов, обычно в них рассматривается вопрос о том, как выполнить анализ на одной из описанных нами платформ. Что касается более общей обработки графов, рекомендуем книгу [Deshpande and Gupta 2018]. Анализ графов – предмет подробного обзора [Yan et al. 2017]. Также отличным источником может служить обзор [Larriba-Pey et al. 2014]. Реальные проблемы обработки графов обсуждаются в работе [Lumsdaine et al. 2007]. Работа [McCune et al. 2015] – хороший обзор ориентированных на вершины систем.

Характеристики графов, в частности асимметричное распределение степеней вершин, играют важную роль в обработке графов. Эта тема обсуждается в работе [Newman et al. 2002]. Важный первый шаг параллельной и распределенной обработки графа – его разбиение, которое занимает большую часть времени обработки [Verma et al. 2017] и с точки зрения вычислений обходится дорого [Andreev and Racke 2006]. Методы разбиения графов можно отнести к двум классам: вершинно-непересекающиеся (реберное разрезание) и реберно-непересекающиеся (вершинное разрезание). Основным алгоритмом, принадлежащим первому классу, является METIS [Karypis and Kumar 1995], его вычислительная сложность проанализирована в работе [McCune et al. 2015]. Еще одну возможность разбиения вершин открывает хеширование. Этот метод дает сбалансированное распределение вершин, но плохо справляется со степенными графами. Были предложены обобщения METIS на этот случай [Abou-Rjeili and Karypis 2006]. Альтернативным подходом является алгоритм распространения меток ([Ugander and Backstrom 2013]). В работе [Wang et al. 2014] предложено также комбинирование METIS с распространением меток путем включения последнего в этап огрубления METIS. Еще один вариант – начать с несбалансированного разбиения и постепенно продвигаться к балансу [Ugander and Backstrom 2013]. Для реберно-непересекающегося разбиения хеширование также возможно. В системе PowerLyra [Chen et al. 2015] реализована комбинация вершинно-непересекающегося и реберно-непересекающегося разбиений.

Предлагалось использовать для обработки графов каркас MapReduce [Cohen 2009, Kiveris et al. 2014, Rastogi et al. 2013, Zhu et al. 2017] и его модификации, обладающие лучшей масштабируемостью [Qin et al. 2014]. Система HaLoop ([Bu et al. 2010, 2012]) – адаптация MapReduce специально для анализа графов. GraphX ([Gonzalez et al. 2014]) – основанная на Spark система, исповедующая подход MapReduce.

Приведенная в разделе 10.4.3 классификация систем, специально созданных для анализа графов, основана на работе [Han 2015, Corbett et al. 2013]. Пошагово-синхронная параллельная модель вычислений (BSP) предложена в работе [Valiant 1990]. К ориентированным на вершины BSP-системам относятся Pregel [Malewicz et al. 2010] и ее аналоги с открытым исходным кодом Apache Giraph [Apache], GPS [Salihoglu and Widom 2013], Mizan [Khayyat et al. 2013], LFGraph [Hoque and Gupta 2013], Pregelix [Bu et al. 2014] и Trinity [Shao et al. 2013]. Оптимизации системного уровня для работы с асимметричными данными осуждаются в работах [Lugowski et al. 2012, Salihoglu and Widom 2013, Gonzalez et al. 2012]. Некоторые алгоритмические оптимизации описаны в работе [Salihoglu and Widom 2014]. К ориентированным на вершины асинхронным системам относятся GRACE [Wang et al. 2013] и GiraphUC [Han and Daudjee 2015]. Основной пример ориентированной на вершины системы сбора–обработки–распространения является GraphLab [Low et al. 2012, 2010]. В основе систем Blogel [Yan et al. 2014] и Giraph++ [Tian et al. 2013] лежит ориентированная на разделы модель BSP. X-Stream [Roy et al. 2013] – на данный момент единственная ориентированная на ребра BSP-система.

Озера данных – новая тема, поэтому пока ей посвящено немного технических книг. Книга [Pasupuleti and Purra 2015] является неплохим введением в архитектуры озер данных с акцентом на руководство данными, безопасность и качество данных. Полезную информацию можно также найти в технических описаниях (например, [Hortonworks 2014]) от компаний, которые поставляют компоненты и службы для озер данных. Часть проблем, с которыми сталкиваются озера данных, связана с интеграцией больших данных. Работа [Dong and Srivastava 2015] представляет собой отличный обзор недавних методов интеграции больших данных. Более широкое рассмотрение интеграции больших данных, включая и веб-данные, – тема работы [Dong and Srivastava 2015]. В этом контексте работа [Coletta et al. 2012] содержит предложения по совместному использованию методов машинного обучения, сопоставления и кластеризации для решения проблем озер данных.

Упражнения

Задача 10.1. Сравните различные подходы к проектированию системы хранения в терминах масштабируемости, простоты использования (внесение данных и т. д.), архитектуры (с разделением и без разделения ресурсов и т. д.), согласования схем, отказоустойчивости, управления метаданными.

Задача 10.2 (*). Рассмотрите программный стек управления большими данными (рис. 10.1) и сравните его со стеком традиционной реляционной СУБД, например показанным на рис. 1.9. В частности, обсудите основные различия в управлении системой хранения.

Задача 10.3 (*). На уровне распределенного хранения стека управления большими данными данные, как правило, хранятся в виде файлов или объектов. Обсудите, когда выгоднее использовать объекты, а когда файлы, исходя

из характеристик данных, например объекты малого или большого размера, большое число объектов, похожие записи, а также требований к приложению: простота перемещения данных между машинами, масштабируемость, отказоустойчивость.

Задача 10.4 (*). В распределенной файловой системе типа GFS или HDFS файлы разбиваются на части фиксированного размера, называемые порциями. Объясните различия между порцией и горизонтальной секцией, определенной в главе 2.

Задача 10.5. Рассмотрите различные реализации эквисоединения в MapReduce, описанные в разделе 10.2.1.3. Сравните широковещательное соединение и соединение с повторным разбиением с точки зрения общности и стоимости тасования.

Задача 10.6 (*). В разделе 10.2.1.1 описано, как модуль комбинатора используется для уменьшения стоимости тасования.

a) Напишите псевдокод такого комбинатора для SQL-запроса из примера 10.1.

b) Опишите, как он мог бы уменьшить стоимость тасования.

Задача 10.7. Рассмотрите реализацию оператора тета-соединения в MapReduce и его поток данных, показанный на рис. 10.10. Обсудите, как такой поток данных влияет на производительность эквисоединения (когда θ совпадает с =).

Задача 10.8. Рассмотрите реализацию в Spark алгоритма кластеризации методом k средних, представленного в примере 10.3. Опишите, какие шаги алгоритма приводят к тасованию данных между исполнителями.

Задача 10.9. В примере 10.4 мы обсудили вычисление PageRank. В варианте, называемом «персонализированный PageRank», вычисляется ценность страницы относительно выбранного пользователем набора страниц. Для этого большая важность придается ребрам в окрестности некоторых страниц, указанных пользователем. В этой задаче мы предположим, что этот набор страниц состоит из единственной страницы, называемой *источником*. Именно относительно этой страницы производится вычисление. От обычного PageRank эта величина отличается в следующих отношениях:

- напомним, что когда в алгоритме PageRank случайное блуждание приводит на некоторую страницу, дальше с вероятностью *d* производится переход на случайную страницу в графе. В персонализированном алгоритме PageRank эта страница не случайная, а всегда совпадает с источником, т. е. с вероятностью *d* происходит возврат назад к *источнику*;
- на этапе инициализации вычисления, вместо того чтобы присваивать равные значения PageRank всем вершинам графа, источнику присваивается ранг 1, а остальным страницам – ранг 0.

Вычислите (вручную) персонализированный PageRank графа, изображенного на рис. 10.21.

Задача 10.10 (**). Реализуйте алгоритм персонализированного PageRank, описанный в задаче 10.9, с помощью каркаса MapReduce (используйте Hadoop).

Задача 10.11 (**). Реализуйте алгоритм персонализированного PageRank, описанный в задаче 10.9, с помощью Spark.

Задача 10.12 (**). Рассмотрим СОПД, описанную в разделе 10.3. Опишите алгоритм, реализующий семантику доставки не меньше одного раза.

Задача 10.13 (**). Рассмотрим СПД, описанную в разделе 10.3.

- **a)** Спроектируйте внутриоператорную параллельную версию потокового оператора фильтрации.
- **b)** Спроектируйте внутриоператорную параллельную версию оператора агрегирования. Указание: в отличие от оператора фильтрации, оператор агрегирования имеет внутреннее состояние. При его реализации необходимо принять во внимание нечто такое, что было неважно в операторе фильтрации (не имеющем состояния). Что именно? Как данные распределяются между экземплярами оператора? Что следует сделать на выходе предыдущего оператора для гарантии того, что каждый потоковый кортеж попадет нужному экземпляру?

Задача 10.14 (**). Спроектируйте оператор соединения двух потоков со скользящим окном. Является ли он детерминированным? Если нет, то почему? Можете ли вы предложить альтернативный проект оператора, который гарантировал бы детерминированность независимо от относительной скорости и чередования входных потоков?

Задача 10.15. Рассмотрим ориентированную на вершины модель программирования для обработки графов (см. раздел 10.4.3). Сравните модели вычислений BSP и GAS с точки зрения:

- **a)** общности и выразительности алгоритмов на графах и
- **b)** оптимизации производительности.

Задача 10.16 (**). Реализуйте алгоритм персонализированного PageRank, описанный в задаче 10.9, с помощью ориентированной на вершины модели BSP.

Задача 10.17 (*). В примере 10.8 описан итеративный основанный на распространении меток алгоритм нахождения компонент связности входного графа в ориентированной на вершины модели программирования. Рассмотрим потоковое приложение, в котором каждый входящий кортеж в потоке представляет неориентированное ребро входного графа. Спроектируйте инкрементный алгоритм, который находит компоненты связности графа, образованного поступающими из потока ребрами.

Задача 10.18 (*). Рассмотрим жадную эвристику размещения ребер, определенную в разделе 10.4.10. Известно, что эта эвристика дает несбалансированную нагрузку, если поток ребер подается злонамеренным противником в заведомо неудобном порядке.

- **a)** Упорядочите вершины на рис. 10.23, так чтобы описанная эвристика создавала максимально несбалансированную нагрузку, т. е. помещала все ребра в один раздел.
- **b)** Предложите стратегию, которая уменьшала бы несбалансированность нагрузки при таком неудачном упорядочении потока.

Задача 10.19 (*). Озеро данных напоминает хранилище данных (см. главу 7), но только для неструктурированных бессхемных данных, например хранящихся в HDFS. Рассмотрим систему больших данных Spark, которая предоставляет доступ на языке SQL (посредством SparkSQL) к HDFS-данным и данным из многих других источников. Достаточно ли Spark для построения озера данных? Какой функциональности не хватает?

Задача 10.20 (**). В последних версиях параллельных СУБД, используемых в современных хранилищах данных, добавлена поддержка внешних таблиц (см., например, описание Polybase в главе 11), которые устанавливают соответствие с HDFS-файлами и могут использоваться наряду со стандартными реляционными таблицами в SQL-запросах. С другой стороны, озера данных предоставляют доступ к внешним источникам данных, например SQL, NoSQL и другим, с помощью оберток, например соединителей Spark. Сравните оба подхода с точки зрения интеграции данных. Назовите сходства и различия.

Глава 11

NoSQL, NewSQL и полихранилища

Для управления данными в облаке всегда можно положиться на реляционную СУБД. У всех реляционных СУБД имеются распределенные версии, и большинство работает в облаке. Однако эти системы критиковали за принятый в них универсальный подход. Несмотря на то что они способны интегрировать данные любого вида (в частности, мультимедийные объекты и документы), а также на появление новых функций, все это приводит к утрате производительности, простоты и гибкости для приложений с конкретными и очень жесткими требованиями к производительности. Поэтому отмечалась необходимость в более специализированных СУБД. Например, показано, что столбцовые СУБД, в которых рядом хранятся столбцы, а не строки, как в традиционных строковых реляционных СУБД, на рабочих нагрузках типа OLAP работают более чем на порядок быстрее. Аналогично системы управления потоками данных (см. раздел 10.3) специально проектируются для эффективной работы с потоками данных.

Таким образом, предложено много разных решений для управления данными, специализированных под конкретные виды данных и задач и способных работать на несколько порядков лучше традиционных реляционных СУБД. Примерами новых технологий могут служить распределенные файловые системы и каркасы для параллельной обработки больших данных (см. главу 10).

Важная разновидность новых технологий управления данными получила название NoSQL, т. е. «Not Only SQL» (не только SQL). Она противопоставляется традиционной универсальной реляционной СУБД. NoSQL-системы представляют собой специализированные склады данных, отвечающие требованиям управления данными в вебе и в облаке. Термин «склад данных» (data store) употребляется часто, потому что он весьма общий и включает не только СУБД, но и более простые файловые системы и каталоги. Будучи альтернативой реляционным СУБД, NoSQL-системы поддерживают различные модели данных и языки, отличающиеся от стандартного SQL. Упор в них делается на масштабируемости, отказоустойчивости и доступности, иногда за счет согласованности. Существует несколько типов NoSQL-систем, включая хранилища ключей и значений, документные базы данных, хранилища

с широкими столбцами, графовые базы, а также гибридные системы (многомодельные или NewSQL).

Эти новые технологии породили предложение широкого спектра служб, которые можно использовать для построения облачных информационно емких высокопроизводительных приложений, допускающих масштабирование. Но это также привело к диверсификации интерфейсов к складам данных и к утрате единой парадигмы программирования. Поэтому пользователю очень трудно создать приложения для работы с несколькими складами данных, например распределенной файловой системой, реляционной СУБД и СУБД типа NoSQL. Это стало поводом для проектирования *полихранилищ*, которые предоставляют интегрированный доступ к нескольким облачным складам данных с помощью одного или нескольких языков запросов.

Эта глава организована следующим образом. В разделе 11.1 обсуждаются причины появления NoSQL-систем, в частности теорема CAP, которая описывает компромисс между различными свойствами. Затем мы познакомимся с различными типами NoSQL-систем: хранилищами ключей и значений в разделе 11.2, хранилищами документов в разделе 11.3, хранилищами с широкими столбцами в разделе 11.4, графовыми базами данных в разделе 11.5. В разделе 11.6 представлены гибридные системы, т. е. многомодельные NoSQL-системы и СУБД типа NewSQL. В разделе 11.7 обсуждаются полихранилища.

11.1. Причины появления NoSQL

Существует несколько взаимодополняющих причин появления NoSQL-систем. Самая очевидная – «универсальность» реляционных СУБД, о которой мы говорили выше.

Вторая причина – ограниченная масштабируемость и доступность, заложенная в ранние архитектуры баз данных, развертываемых в облаке. Это традиционная трехъярусная архитектура с балансировщиком нагрузки, веб-серверами приложений, серверами баз данных и клиентами, обращающимися к центру обработки данных (ЦОД). Обычно в ЦОДе используется кластер без разделения ресурсов – самое экономически эффективное решение для облака. Для данного приложения существует единственный сервер баз данных, обычно реляционный, который обеспечивает отказоустойчивость и доступность данных посредством репликации. Когда количество веб-клиентов возрастает, можно сравнительно легко добавить новые веб-серверы, как правило, в виде виртуальных машин, которые возьмут на себя увеличившуюся нагрузку. Но сервер баз данных становится узким местом, а добавление нового сервера означает необходимость репликации всей базы данных, что занимает много времени. В кластере без разделения ресурсов для обеспечения масштабируемости можно было бы воспользоваться параллельной реляционной СУБД. Однако такое решение годится лишь для рабочих нагрузок типа OLAP (с преобладанием чтения) (см. раздел 8.2) и экономически неэффективно, потому что параллельные реляционные СУБД – очень дорогой продукт.

Третья причина появления NoSQL-систем – тот факт, что поддержка сильной согласованности, обеспечиваемая реляционными СУБД посредством ACID-транзакций, вступает в противоречие с масштабируемостью. Поэтому в некоторых NoSQL-системах требование сильной согласованности ослаблено в пользу масштабируемости. В обоснование такого подхода часто приводят знаменитую теорему CAP из теории распределенных систем. Однако это порочная аргументация, потому что теорема CAP не имеет никакого отношения к масштабируемости базы данных, а говорит о согласованности репликации при наличии разделения сети. К тому же вполне возможно обеспечить одновременно сильную согласованность базы данных и масштабируемость, как доказывают некоторые NewSQL-системы (см. раздел 11.6.2).

Теорема CAP утверждает, что распределенный склад данных с репликацией может обеспечить выполнение только двух из следующих трех свойств: (C) согласованность (Consistency), (A) доступность (Availability) и (P) устойчивость к разделению (Partition tolerance). Ни о какой масштабируемости (S) и речи нет. Эти свойства определены следующим образом:

- согласованность: все узлы в каждый момент времени видят одни и те же данные, т. е. любая операция чтения возвращает последнее записанное значение. Это свойство соответствует линеаризуемости (согласованности на уровне отдельных операций), а не сериализуемости (согласованности на уровне групп операций);
- доступность: любая реплика должна отвечать на любой полученный запрос;
- устойчивость к разделению: система продолжает функционировать, несмотря на разделение сети вследствие отказа.

Многие неправильно интерпретируют теорему CAP, считая, что одним из трех свойств нужно пожертвовать. Но лишь в случае разделения сети действительно необходимо выбирать между согласованностью и доступностью.

NoSQL (Not Only SQL) – перегруженный термин, который оставляет большое пространство для интерпретации и определений. Например, его можно применить к ранним иерархическим и сетевым СУБД, или к объектным СУБД, или к СУБД на основе XML. Однако впервые этот термин появился в конце 1990-х годов и обозначал новые склады данных, удовлетворяющие требованиям, предъявляемым к управлению данными в вебе и в облаке. Будучи альтернативами реляционным базам данных, они поддерживают различные модели данных и языки, отличающиеся от стандартного SQL. Акцент в этих системах ставится на масштабируемость, отказоустойчивость и доступность, иногда ценой отказа от согласованности.

В этой главе мы познакомимся с четырьмя основными категориями NoSQL-систем, определяемыми моделью данных, а именно: хранилищами ключей и значений, хранилищами с широкими столбцами, документными и графовыми базами. Мы также рассмотрим гибридные базы данных: многомодельные, сочетающие несколько моделей данных в одной системе, и NewSQL, сочетающие масштабируемость NoSQL с сильной согласованностью реляционных СУБД. Для каждой категории мы приведем пример существующей системы.

11.2. Хранилища ключей и значений

В этой модели все данные представлены в виде пар ключ-значение, причем ключ однозначно определяет значение. Хранилища ключей и значений бессхемные, что обеспечивает значительную гибкость и масштабируемость. Обычно они предоставляют интерфейс в виде функций `put(key, value)`, `value=get(key)`, `delete(key)`.

Расширенное хранилище ключей и значений способно хранить записи, т. е. списки пар атрибут-значение. Первый атрибут называется главным ключом (например, это может быть номер социального страхования) и однозначно идентифицирует запись в коллекции записей, например группе людей. Обычно ключи отсортированы, что открывает возможность для запросов по диапазону и обработки ключей по порядку.

Популярным хранилищем ключей и значений является Amazon DynamoDB, с которым мы и познакомимся ниже.

11.2.1. DynamoDB

Хранилище DynamoDB используется некоторыми из главных служб Amazon, которым необходима высокая доступность и доступ к данным по ключу, например службы для поддержки корзин покупок, списков покупателей, предпочтений пользователей и товарных каталогов. Для обеспечения масштабируемости и доступности DynamoDB жертвует согласованностью при некоторых отказах и применяет синтез хорошо известных по P2P-системам методов (см. главу 9) в кластере без разделения ресурсов.

В DynamoDB данные хранятся в таблицах, представляющих собой коллекции отдельных элементов. Каждый элемент является списком пар атрибут-значение. Значением атрибута может быть скаляр или структура в формате JSON. Элементы аналогичны строкам реляционной таблицы, а атрибуты – столбцам. Но поскольку атрибуты самоописываемые, реляционная схема не нужна. Кроме того, элементы могут быть гетерогенными, т. е. иметь разный набор атрибутов.

В оригинальном проекте DynamoDB было две распределенные DHT-таблицы (см. раздел 9.1.2). Первичный ключ (первый атрибут) хешируется в разные секции, что позволяет эффективно выполнять операции чтения и записи по ключу, а также обеспечивает сбалансированную нагрузку. В более поздних версиях DynamoDB поддерживает составные первичные ключи из двух атрибутов. Первый атрибут является ключом хеширования и не обязан быть уникальным. Второй атрибут – ключ диапазона, он позволяет выполнять операции по диапазону в секции, соответствующей ключу хеширования. Для доступа к таблицам базы данных DynamoDB предоставляет API на языке Java со следующими операциями:

- `PutItem`, `UpdateItem`, `DeleteItem`: добавление, обновление и удаление элемента таблицы по первичному ключу (либо первичному ключу хеширования, либо составному первичному ключу);
- `GetItem`: возвращает элемент таблицы с указанным первичным ключом;

- `BatchGetItem`: возвращает элементы таблицы с одинаковым первичным ключом, но в нескольких таблицах;
- `Scan`: возвращает все элементы таблицы;
- запрос по диапазону: возвращает все элементы диапазона с данным ключом хеширования и данным ключом диапазона;
- индексированный запрос: возвращает все элементы с данным значением индексированного атрибута.

Пример 11.1. Рассмотрим таблицу Forum_Thread на рис. 11.1. Она состоит из однородных элементов с четырьмя атрибутами: Forum, Subject, Date of last post и Tags. Таблица имеет составной ключ, составленный из ключа хеширования (Forum) и ключа диапазона (Subject). Запрос по первичному ключу

```
GetItem(Forum="EC2," Subject="xyz")
```

возвращает последний элемент. Запрос по диапазону

```
Query(Forum="S3," Subject >"ac")
```

возвращает второй и третий элементы. ♦

Таблица: Forum_Thread

Forum	Subject	Date of last post	Tags
“S3”	“abc”	“2017 …”	“a” “b”
“S3”	“acd”	“2017 …”	“c”
“S3”	“cbd”	“2017 …”	“d” “e”
“RDS”	“xyz”	“2017 …”	“f”
“EC2”	“abc”	“2017 …”	“a” “e”
“EC2”	“xyz”	“2017 …”	“f”

Ключ хеширования — Ключ диапазона

Рис. 11.1 ❖ Пример таблицы DynamoDB

DynamoDB строит неупорядоченный хеш-индекс по ключу хеширования, т. е. DHT, и отсортированный индекс по ключу диапазона. Кроме того, DynamoDB предлагает два вида вторичных индексов для быстрого доступа по неключевым атрибутам: локальные для выборки элементов из одной секции, т. е. элементов с одинаковым значением ключа хеширования, и глобальные для выборки элементов из всей таблицы DynamoDB.

Данные секционируются и реплицируются на несколько узлов кластера в нескольких ЦОДах, что обеспечивает балансировку нагрузки и высокую доступность. Для секционирования используется согласованное хеширование – популярная схема, применяемая в DHT с кольцевой геометрией, например Chord (см. раздел 9.1.2). DHT представлена в виде одномерного кольцевого пространства идентификаторов, и каждому узлу системы присваивается

случайное значение из этого пространства, представляющее его позицию в кольце. Для отнесения элемента к узлу вычисляется хеш ключа элемента, в результате чего определяется позиция в кольце, а затем находится первый по часовой стрелке узел, позиция которого больше, чем позиция элемента. Таким образом, каждый узел отвечает за интервал кольца между своим предшественником и собой. Главное достоинство согласованного хеширования заключается в том, что добавление (присоединение) и удаление (выход или отказ) узлов влияют только на ближайшего соседа и больше ни на какие узлы.

В DynamoDB согласованное хеширование используется также для обеспечения высокой доступности благодаря тому, что каждый элемент реплицируется на n узлах, где n – настраиваемый параметр системы. Каждому элементу сопоставляется узел-координатор, как описано выше, и элемент реплицируется на $n - 1$ следующих по часовой стрелке узлах. Таким образом, каждый узел отвечает за интервал кольца между своим n-м предшественником и собой.

Пример 11.2. На рис. 11.2 показано кольцо с 6 узлами, поименованными в соответствии с позицией (хеш-значением). Узел B отвечает за интервал хеш-значений (A, B], а узел A – за интервал (F, A]. Операция `put(c,v)` возвращает хеш-значение ключа *c* между A и B, поэтому узел B становится ответственным за этот элемент. Кроме того, если параметр репликации $n = 3$, то элемент будет реплицирован на узлах C и D. Таким образом, в узле D хранятся все элементы с ключами, попадающими в интервалы (A, B], (B, C] и (C, D].

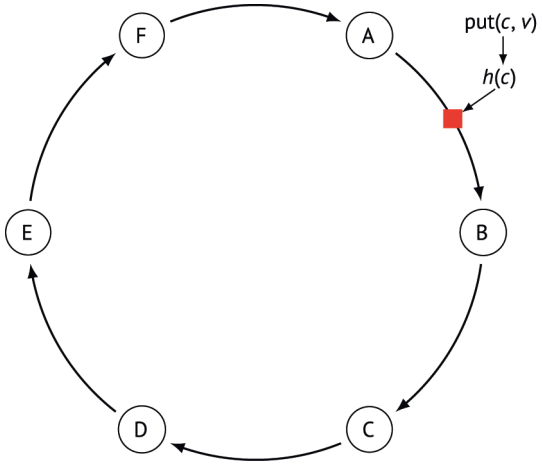


Рис. 11.2 ❖ Согласованное хеширование в DynamoDB. Узел B отвечает за интервал хеш-значений (A, B]. Поэтому элементу (c, v) назначается узел B

DynamoDB жертвует сильной согласованностью данных ради масштабируемости и доступности, но располагает различными способами управления согласованностью. Она обеспечивает согласованность реплик в конечном счете (см. раздел 6.1.1), что достигается с помощью асинхронного протокола

распространения обновлений и распределенного протокола обнаружения отказов на основе сплетен.

Согласованностью записи в конкурентной многопользовательской среде можно управлять с помощью условных операций записи. По умолчанию операции записи (`PutItem`, `UpdateItem`, `DeleteItem`) перезаписывают существующий элемент с заданным первичным ключом. В случае условной записи имеется условие на атрибуты элемента, при выполнении которого операция завершается успешно. Например, условием успеха `PutItem` является отсутствие элемента с таким же первичным ключом. Поэтому условная запись полезна в случае конкурентных обновлений.

DynamoDB поддерживает сильно согласованное и согласованное в конечном счете чтение. По умолчанию операции чтения согласованы в конечном счете, т. е. могут возвращать не последнее значение асинхронно реплицируемых данных. Сильно согласованная операция чтения возвращает самое актуальное значение данных, что может оказаться невозможным в случае отказа сети, поскольку обновление распространилось не на все реплики.

11.2.2. Другие хранилища ключей и значений

Из других популярных хранилищ ключей и значений отметим Cassandra, Memcached, Riak, Redis, Amazon SimpleDB и Oracle NoSQL Database. Многие системы поддерживают расширения, чтобы обеспечить плавный переход к хранилищам с широкими столбцами и документным хранилищам, которые мы обсудим далее.

11.3. Документные хранилища

Документные хранилища представляют собой усовершенствованные хранилища ключей и значений, в которых ключам соответствуют значения типа документа, например в формате JSON, YAML или XML. Документы, как правило, группируются в коллекции, играющие ту же роль, что реляционные таблицы. Документы самоописываемые, т. е. помимо данных содержат и метаданные (например, разметку в XML или имена полей в JSON-объектах). Документы в одной коллекции могут иметь разную структуру. Кроме того, документы являются иерархическими, т. е. могут содержать вложенные конструкции, например вложенные объекты и массивы в случае JSON. Поэтому для моделирования базы данных с помощью документов нужно меньше коллекций, чем при использовании плоских реляционных таблиц, что позволяет избежать дорогостоящих операций соединения.

Помимо простого интерфейса для извлечения документов по ключу, документное хранилище предлагает API или язык запросов для поиска документов по содержимому. Документные хранилища упрощают обработку изменений и необязательных значений, а также отображение документов на объекты программы. Из-за этого они привлекают разработчиков современ-

ных постоянно изменяющихся веб-приложений, для которых очень важна быстрота развертывания.

Популярным документным хранилищем является MongoDB, мы рассмотрим его ниже.

11.3.1. MongoDB

MongoDB – система с открытым исходным кодом, написанная на C++. Она предлагает документную модель данных на основе JSON, гибкие схемы, высокую доступность, отказоустойчивость и масштабируемость в кластерах без разделения ресурсов.

В MongoDB документы хранятся в формате BSON (бинарный JSON) – сериализованном двоично-кодированном представлении JSON, включающем дополнительные типы binary, int, long и floating. BSON-документ состоит из одного или нескольких полей, каждое поле имеет имя и содержит значение определенного типа, в т. ч. массив, двоичные данные и поддокументы. Каждый документ имеет уникальный идентификатор – первое поле типа ObjectId, которое автоматически генерирует MongoDB.

Все документы с похожей структурой организованы в коллекции, аналогичные реляционным таблицам, а поля документов аналогичны столбцам таблицы. Однако документы, входящие в одну коллекцию, могут иметь разную структуру, поскольку обязательной схемы не существует. MongoDB предлагает развитый язык запросов, позволяющий обновлять и извлекать данные из BSON-документов с помощью функций, выраженных на JSON. Представление запросов в формате JSON позволяет унифицировать способ хранения и манипулирования данными. Для работы с языком запросов имеется API на разных языках программирования, в т. ч. Java, PHP, JavaScript и Scala. Поскольку запросы реализованы в виде методов или функций с применением API для конкретного языка программирования, интеграция с прикладными программами оказывается простой и естественной.

MongoDB поддерживает различные виды запросов для вставки, обновления, удаления и извлечения документов. Запрос может возвращать документы, отдельные поля документов или сложные агрегаты значений из многих документов. Запросы могут также включать определенные пользователями функции на языке JavaScript. В общем виде запрос выглядит так:

```
db.collection.function (выражение JSON),
```

где *db* – глобальная переменная, описывающая подключение к базе данных, а *function* – операция базы данных, применяемая к коллекции. Произвольное выражение JSON используется как критерий отбора данных. Поддерживаются следующие виды запросов:

- операции вставки, удаления и обновления документов. Для операций удаления и обновления выражение JSON задает критерий отбора;
- запросы с точным совпадением возвращают результаты, полученные сравнением на равенство с некоторым полем документа, обычно первичным ключом;

- запросы по диапазону возвращают результаты из документов, поле которых принадлежит заданному диапазону;
- геопространственные запросы возвращают результаты на основе близости, пересечения и включения географических объектов, например точки, прямой линии, окружности или многоугольника, представленных в формате GeoJSON;
- текстовые запросы возвращают результаты, полученные применением булевых операторов к текстовым аргументам и упорядоченные по релевантности;
- запросы с агрегированием возвращают результаты агрегирования коллекции с помощью таких операторов, как count, min, max и average. Кроме того, документы из двух коллекций можно комбинировать с помощью операции левого внешнего соединения.

Пример 11.3. Рассмотрим коллекцию posts на рис. 11.3. Каждый элемент коллекции однозначно идентифицируется своим ключом типа ObjectId (генерируемым MongoDB), а значением ключа является JSON-объект с вложенными массивами, например тегов и комментариев. Ниже приведено несколько примеров операции обновления:

```
db.posts.insert(author:“alex,” title:“No Free Lunch”)
db.posts.update(author:“alex,” $set:age:30)
db.posts.update(author:“alex,” $push:tags:“music”)
```

где $set (записывает в поле указанное значение) и $push (добавляет указанное значение в конец массива) – команды MongoDB внутри JSON.

_id: ObjectId(“abc”)	author: “alex”, title: “No Free Lunch”, text: “This is . . .”, tags: [“business”,“ramblings”], comments: [{who:“jane”, what:“I agree.”},{who:“joe”, what:“No. . .”}]
_id: ObjectId(“abd”)	Сообщение от X
_id: ObjectId(“acd”)	Сообщение от Y

Уникальный ключ, сгенерированный MongoDB — Значение = JSON-объект с вложенными массивами

Рис. 11.3 ❖ Пример коллекции posts в MongoDB

Запросы

```
db.posts.find(author:“alex”)
db.posts.find(comments.who:“jane”)
```

являются запросами с точным совпадением. Первый возвращает все сообщения от Алекса, а второй – все сообщения, прокомментированные Джейн. ♦

Для эффективного доступа к данным MongoDB поддерживает разного рода вторичные индексы, которые можно построить по любому полю документа, в т. ч. по полям внутри массива. Перечислим виды индексов:

- уникальные индексы, для которых значение индексируемого поля должно быть уникальным;
- составные индексы, построенные по нескольким полям;
- индексы по полям массива, в которых каждому элементу массива соответствует отдельная запись индекса;
- TTL-индексы, которые автоматически уничтожают документы по истечении заданного времени жизни (TTL);
- геопространственные индексы для оптимизации запросов с учетом близости, пересечения и включения географических объектов (точки, прямой, окружности или многоугольника);
- частичные индексы, построенные над подмножеством документов, удовлетворяющих заданному пользователем условию;
- разреженные индексы, построенные только по документам, содержащим указанное поле;
- полнотекстовые индексы, в которых используются лингвистические правила конкретного языка для оптимизации запросов по текстовому полю.

Для горизонтального масштабирования в кластерах без разделения ресурсов MongoDB поддерживает различные виды секционирования данных (шардинга в терминологии MongoDB): на основе хеширования, на основе диапазонов или с учетом местоположения (когда пользователь задает диапазоны ключей и ассоциированные с ними узлы). Высокая доступность достигается за счет варианта репликации главной копии – наборов реплик – с помощью асинхронного распространения обновлений. Если главный узел выходит из строя, то одна из реплик становится новым главным узлом и продолжает принимать операции обновления. Кластер MongoDB состоит из: шардов (секций данных), каждый из которых может быть единицей репликации (набором реплик), mongo-узлов, играющих роль процессоров запросов от клиентских приложений кластеру, и конфигурационных серверов, на которых хранятся метаданные и конфигурационные параметры кластера. Приложения могут при желании читать из вторичных реплик, понимая, что данные согласованы лишь в конечном счете.

Недавно в MongoDB появилась поддержка ACID-транзакций с несколькими документами в дополнение к однодокументным транзакциям. Это достигается с помощью изоляции моментальных снимков. В одной транзакции можно записать одно или несколько полей документа, включая обновление нескольких поддокументов и элементов массива. В многодокументных транзакциях может участвовать несколько коллекций, баз данных, документов и шардов.

MongoDB также поддерживает параметр *заинтересованность в записи* (write concern), который позволяет задать уровень гарантий для уведомления об успешности операции записи, иначе говоря, желаемый компромисс между производительностью и сохранением в репликах базы данных. Существует четыре уровня гарантий, перечислим их в порядке от самой слабой к самой сильной: unacknowledged (никаких гарантий), acknowledged (произведена запись на диск), journaled (операция записи зарегистрирована в журнале), replica acknowledged (запись распространена на все реплики).

Архитектура MongoDB совместима с программным стеком управления большими данными (см. рис. 10.1). Она допускает взаимозаменяемые подсистемы хранения, например HDFS или в оперативной памяти, отвечающие потребностям конкретных приложений, имеет интерфейсы с такими системами обработки больших данных, как MapReduce и Spark, и поддерживает сторонние инструменты для аналитики, интернета вещей, мобильных приложений и т. д. (см. рис. 11.4). Для ускорения операций с базой данных MongoDB активно использует оперативную память, а также собственную подсистему хранения (WiredTiger) со сжатием.

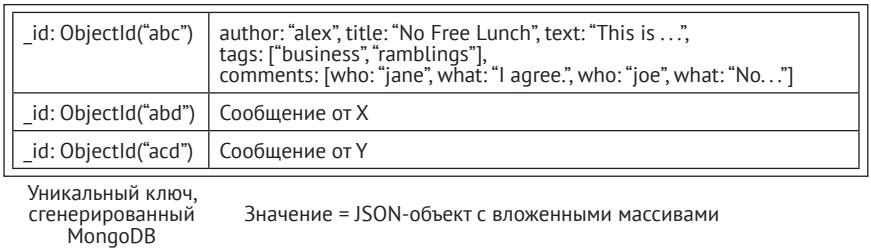


Рис. 11.4 ❖ Архитектура MongoDB

11.3.2. Другие документные хранилища

Из других популярных документных хранилищ отметим AsterixDB, Couchbase, CouchDB и RavenDB, все они поддерживают модель данных JSON в масштабируемой архитектуре кластера без разделения ресурсов. Однако AsterixDB и Couchbase поддерживают диалект SQL++, элегантное расширение SQL с добавлением нескольких простых возможностей для опроса данных в формате JSON. Диалект Couchbase SQL++ называется N1QL (Non-first normal form Query, произносится «никель»). Couchbase поддерживает ограниченные транзакции (атомарная запись документов) и позволяет пожертвовать некоторыми свойствами в обмен на производительность, например ослабить свойство долговечности, допустив подтверждение операций в памяти с последующей асинхронной записью на диск. Помимо сервера, на который возложено выполнение запросов и транзакций, на платформе Couchbase имеется аналитическая служба, которая анализирует данные в оперативном режиме не в ущерб производительности транзакций и не требуя какой-то специальной модели данных или ETL (результата). Это вариант гибридной транзакционно-аналитической обработки (Hybrid Transaction and Analytics Processing – HTAP) для NoSQL (см. введение в HTAP в разделе 11.6.2). Реализация этой аналитической службы основана на подсистеме хранения и параллельном процессоре запросов AsterixDB – высокопроизводительном хранилище JSON-документов, в котором реализована комбинация методов из параллельных и документных баз данных.

11.4. Хранилища с широкими столбцами

Хранилища с широкими столбцами сочетают некоторые удобные средства реляционных баз данных (например, представление данных в виде таблиц) с гибкостью хранилищ ключей и значений (например, хранение бессхемных данных в столбцах). Каждая строка таблицы с широкими столбцами однозначно определяется ключом и состоит из нескольких именованных столбцов. Но в отличие от реляционной таблицы, в столбцах которой можно хранить только атомарные значения, здесь столбец может быть широким и содержать несколько пар ключ-значение.

Хранилища с широкими столбцами расширяют интерфейс хранилищ ключей и значений, добавляя декларативные конструкции для задания запросов к семействам столбцов с последовательным просмотром, точным совпадением и по диапазону. Обычно для таких конструкций предоставляется API, который можно использовать из языков программирования. Некоторые системы предоставляют также SQL-подобный язык запросов, как, например, Cassandra Query Language (CQL).

Прародителем всех хранилищ с широкими столбцами была система Google Bigtable, с которой мы познакомимся ниже.

11.4.1. Bigtable

Bigtable – хранилище с широкими столбцами для кластеров без разделения ресурсов. В Bigtable используется файловая система Google (Google File System – GFS) для хранения структурированных данных в распределенных файлах, обеспечивающих отказоустойчивость и доступность (см. раздел 10.1 о блочных распределенных файловых системах). Предлагается также вариант динамического секционирования данных для обеспечения масштабируемости. Как и GFS, оно используется в таких популярных приложениях Google, как Google Earth, Google Analytics и Google+.

Bigtable поддерживает простую модель данных, напоминающую реляционную, с многозначными атрибутами, снабженными временными метками. Мы кратко опишем эту модель, поскольку она лежит в основе реализации Bigtable, объединяющей некоторые аспекты строковых и столбцовых СУБД. Для совместимости с уже знакомыми нам понятиями мы представим модель данных Bigtable как слегка расширенную реляционную модель[1].

Экземпляр Bigtable – это коллекция пар (ключ, значение), где ключ идентифицирует строку, а значением является множество столбцов, организованных в семейства. Bigtable сортирует хранимые данные по ключу, что помогает собрать строки, принадлежащие одному диапазону, в одном узле кластере.

Каждая строка Bigtable однозначно идентифицируется *ключом строки*, который может быть произвольной строкой (размером до 64 КБ в оригиналь-

[1] В оригинальном предложении Bigtable определено как многомерное отображение, индексированное по ключу строки, ключу столбца и временной метке, в котором каждая ячейка отображения хранит единственное значение (строку).

ной системе). Таким образом, ключ строки – аналог одноатрибутного ключа отношения. Данные в Bigtable всегда отсортированы по ключу строки. Строка может содержать несколько *семейств столбцов*, являющихся единицей контроля доступа и хранения. В семейство входят столбцы одного и того же типа. Для создания Bigtable достаточно задать имя таблицы и имена семейств столбцов. Впоследствии в семейство можно динамически добавлять новые столбцы (того же типа).

Для доступа к данным в Bigtable необходимо идентифицировать столбцы внутри семейств столбцов с помощью *ключей столбцов*. Ключ столбца представляет собой полное имя вида **имя-семейства-столбцов:имя-столбца**. Имя семейства столбцов аналогично имени атрибута отношения. Имя столбца аналогично значению атрибута отношения, но используется как имя, составляющее часть ключа столбца для представления одного элемента. Получается эквивалент многозначных атрибутов отношения. Кроме того, данные, идентифицируемые ключом столбца внутри строки, могут иметь несколько версий, идентифицируемых временной меткой (64-разрядное целое).

Пример 11.4. На рис. 11.5 показан пример Bigtable с 3 семействами столбцов и 2 строками, представленный в реляционном стиле. Семейства столбцов Name и EMail состоят из разнородных столбцов. Для доступа к значению столбца нужно указать ключи строки и столбца, например: ключ строки = “111” и ключ столбца = “Email:gmail.com”. В ответ получим “am@gmail.com”. ♦

Ключ строки	Name	Email	Web page
100	“Prefix”: “Dr.” “Last”: “Dobb”	“email: gmail.com”: “dobb@gmail.com”	<!DOCTYPE html PUBLIC. . . >
101	“First”: “Alice” “Last”: “Martin”	“email: gmail.com”: “amartin@gmail.com” “email: free.fr”: “amartin@free.fr”	<!DOCTYPE html PUBLIC. . . >

Рис. 11.5 ❖ Bigtable с 3 семействами столбцов и 2 строками

Bigtable предоставляет базовый API для определения и манипулирования таблицами из языка программирования, например C++. Кроме того, предоставляются функции для изменения таблицы и метаданных семейства столбцов, например прав доступа. API включает различные операторы для записи и обновления значений и для обхода подмножеств данных, порождаемых оператором сканирования. Существуют различные способы ограничить множество строк, столбцов и временных меток, порождаемых в результате сканирования, как в реляционном операторе выборки. Однако нет сложных операторов вроде соединения или объединения, их нужно программировать с помощью оператора сканирования.

Атомарные транзакции поддерживаются только для обновления одной строки. Если требуется более сложное обновление, охватывающее несколько строк, то программист должен написать код для управления атомарностью, применяя интерфейс пакетной записи строк с разными ключами на стороне клиента.

Для сохранения таблицы в GFS Bigtable применяет секционирование по диапазону ключей строк. Каждая таблица разбивается на секции, называемые *таблетами* (tablet), соответствующие диапазонам строк. Секционирование динамическое, вначале имеется одна таблета (охватывающая всю таблицу), а затем по мере роста таблицы производится разбиение на несколько таблет. Чтобы найти таблеты с пользовательскими данными в GFS, Bigtable использует таблицу метаданных, которая сама секционирована на таблеты, и единственная корневая таблета хранится на главном сервере, аналогичном главному серверу GFS. Помимо использования GFS ради масштабируемости и доступности, Bigtable применяет различные методы для оптимизации доступа к данным и минимизации количества обращений к диску, например сжатие семейств столбцов, группировка семейств столбцов с высокой локальностью доступа и агрессивное кеширование метаданных на стороне клиентов.

Bigtable опирается на высокодоступную службу распределенной блокировки Chubby. Эта служба состоит из пяти активных реплик, одна из которых избирается главной и обслуживает запросы. Bigtable использует Chubby для решения нескольких задач: гарантировать, что в каждый момент времени имеется не более одного активного главного узла; хранить начальное местоположение данных Bigtable; находить серверы таблет и завершать процедуру удаления сервера таблет; хранить схемы Bigtable. Если Chubby оказывается недоступной в течение длительного времени, то и Bigtable станет недоступной.

11.4.2. Другие хранилища с широкими столбцами

Существуют популярные реализации Bigtable с открытым исходным кодом, например Hadoop Hbase – широко известная реализация на Java, работающая поверх HDFS, и Cassandra, сочетающая идеи Bigtable и DynamoDB.

11.5. Графовые СУБД

Мы познакомились с анализом графов в главе 10 и видели, что иногда весь граф приходится обрабатывать несколько раз до достижения сходимости. Что касается графовых СУБД, то они поддерживают неитеративные запросы и могут обращаться только к части графа, эффективно используя индексы. В графовых базах данные представлены и хранятся в виде графов, что упрощает формулировку и ускоряет обработку запросов к графам, например вычисление кратчайшего пути между двумя вершинами. Это гораздо эффективнее, чем хранить граф в реляционной базе, где данные находятся в разных таблицах, а для запросов нужно выполнять дорогостоящие операции соединения. Графовые СУБД обычно предоставляют мощный язык запросов к графам. Они стали особенно популярны с распространением информационно емких веб-приложений, таких как социальные сети и рекомендательные системы.

В графовых СУБД можно определить гибкую схему, задав типы и свойства вершин и ребер. Это дает возможность определить индексы для быстрого доступа к вершинам по значению свойства, например по названию города, в дополнение к структурным индексам. Запросы к графам можно выражать с помощью графовых операторов, используя специальный API или декларативный язык запросов.

Выше мы видели, что для горизонтального масштабирования очень больших баз данных в хранилищах ключей и значений, документных хранилищах и хранилищах с широкими столбцами данные распределяются между несколькими узлами кластера. Такое секционирование хорошо работает, потому что речь идет об индивидуальных элементах. Но секционировать графовые данные гораздо труднее, поскольку задача оптимального разбиения графа NP-трудная (см. раздел 10.1). Напомним, в частности, что в разных разделах могут оказаться элементы, соединенные ребрами. Проход по таким ребрам влечет затраты на коммуникацию, что снижает производительность операций обхода графа. Поэтому количество «межраздельных» ребер следует минимизировать. Однако при этом могут образоваться несбалансированные разделы.

Далее мы проиллюстрируем графовые СУБД на примере Neo4j, популярной системы, развернутой во многих вычислительных центрах.

11.5.1. Neo4j

Neo4j – коммерческая система с открытым исходным кодом, рекламируемая как масштабируемая высокопроизводительная графовая СУБД с естественным форматом хранения графов, предназначенная для работы в кластерах без разделения ресурсов. Она предлагает развитую модель данных с ограничениями целостности, мощный язык запросов Cypher, индексы, ACID-транзакции и поддержку высокой доступности и балансировки нагрузки.

Модель данных основана на ориентированных графах с раздельным хранением ребер (которые называются связями), вершин (узлов) и атрибутов (свойств). У каждого узла может быть сколько угодно свойств в виде пар (атрибут, значение). Связь должна иметь тип, описывающий ее семантику, и направление от одного узла к другому (или к самому себе). У Neo4j есть важное свойство – по связи можно проходить в обоих направлениях с одинаковой производительностью. Это упрощает моделирование графовой СУБД, потому что нет необходимости создавать две разные связи между узлами, если одна вытекает из другой, как в случае взаимной связи (если A – друг B, то B – друг A) или взаимно однозначной связи типа «владеет» (обратную связь можно назвать «принадлежит»).

Обновление графа включает обновление узлов, связей и свойств, которое необходимо производить согласованным образом. Это делается с помощью ACID-транзакций.

Пример 11.5. На рис. 11.6 приведен пример простого графа социальной сети. Связи «является другом» между Бобом и Мэри («Боб является другом Мэри»)

достаточно для представления взаимной связи (надо полагать, что и Мэри является другом Боба). Ее можно было бы представить и другим способом (от Мэри к Бобу). Это упрощает модель графа.

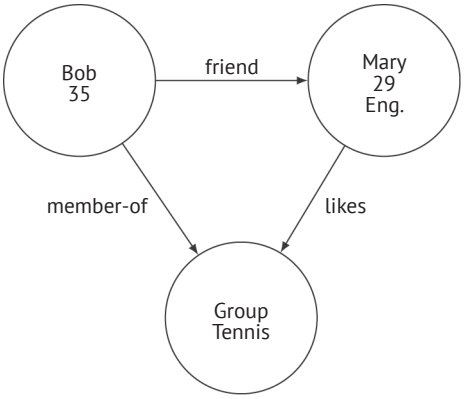


Рис. 11.6 ❖ Примера графа в Neo4j

Следующая транзакция, запрограммированная с помощью Java API, создает узлы Bob и Mary, их свойства и, наконец, связь «является другом» между Бобом и Мэри.

```
Transaction tx = neo.beginTx();
Node n1 = neo.CreateNode();
n1.setProperty("name", "Bob");
n1.setProperty("age", 35);
Node n2 = neo.createNode();
n2.setProperty("name", "Mary");
n1.setProperty("age", 29);
n1.setProperty("job", "engineer");
n1.createRelationshipTo(n2, RelTypes.friend);
tx.Commit();
```

♦

Neo4j не накладывает никакой схемы на граф, что позволяет создавать данные, не вполне понимая заранее, как они будут использоваться. Однако наличие схем в других моделях данных (реляционной, объектной, XML и т. д.) доказало их полезность для поддержания согласованности и эффективной обработки запросов. Поэтому в Neo4j имеется механизм факультативных схем, основанный на понятии метки, и соответствующий язык определения данных. Метки полезны для группировки похожих узлов. Узлу можно сопоставить сколько угодно меток, например: человек, студент и пользователь. Это позволяет Neo4j опрашивать лишь некоторое подмножество графа, например студентов в указанном городе. Метки используются при определении ограничений целостности и индексов. Ограничения целостности можно определять для узлов и связей, например: уникальное свойство узла, существование свойства у узла или существование свойства у связи.

По меткам и комбинациям свойств можно создавать индексы. Они обеспечивают эффективный поиск узлов – важную операцию, которая необходима, чтобы начать обход графа с узлов, описываемых предикатами, которые содержат метки и свойства. Библиотека Neo4j-spatial также предлагает *n*-мерные полигональные индексы для оптимизации геопространственных запросов.

Для запроса к графам и манипулирования данными Neo4j предлагает Java API и язык запросов Cypher. Java API дает программисту на Java доступ к операциям над узлами, связями, свойствами и метками графа с применением ACID-транзакций. Этот API тесно интегрирован с языком программирования.

Cypher – мощный язык запросов графовой СУБД, напоминающий SQL. Его можно использовать для манипулирования данными графа. Например, транзакцию из примера 11.5 можно было бы записать с помощью следующей команды **CREATE**:

```
CREATE (:Person {name:"Bob", age:35}) <- [:FRIEND]
  -(:Person {name:"Mary", age:29, job:"engineer"})
```

Cypher легко использовать благодаря сопоставлению с графовыми образцами: пользователь задает графовый образец, как при рисовании диаграмм, и просит базу найти данные, отвечающие этому образцу. В Cypher имеются фразы **MATCH**, **MERGE**, **WHERE**, **RETURN**, с помощью которых можно манипулировать узловыми переменными (как кортежными переменными в SQL), а также несколько других. Фраза **MATCH** служит для сопоставления с образцом. Узлы обозначаются скобками, а связи – парами минусов и знаком > или <, описывающим направление связи. Пары ключ-значение, задающие свойства узла или связи, записываются в фигурных скобках. Фраза **MERGE** полезна для создания или сравнения графов. Фраза **WHERE** задает предикат для узлов, а фраза **RETURN** – возвращаемые узлы, связи и свойства.

Пример 11.6. Следующий запрос на языке Cypher возвращает всех прямых и непрямых друзей Боба с именами, начинающимися на «M». Фраза **MATCH** определяет рекурсивный образец для связи друг друга с узловыми переменными `bob` и `follower`. Фраза **WHERE** ищет узел с именем «Bob», для чего можно использовать индекс, и выбирает следующие за ним узлы с именами, начинающимися на «M». Фраза **RETURN** возвращает все пары узлов, привязанные к переменным `bob` и `follower`.

```
MATCH bob-[:FRIEND]-> ()-[:FRIEND] -> follower
WHERE bob.name = "Bob" AND follower.name =~ "M.*"
RETURN bob, follower.name
```
♦

Neo4j включает стоимостной компилятор запросов, который порождает оптимизированные планы выполнения Cypher-запросов чтения и обновления. Сначала компилятор производит логическое переписывание запроса, применяя раскрутку вложенности, объединение и упрощение различных его частей. Затем, основываясь на статистической информации об индексе и избирательности меток, компилятор выбирает лучшие методы доступа

для операторов и порождает план выполнения с вложенными итераторами, который выполняется в конвейерном режиме сверху вниз.

Neo4j обеспечивает высокую доступность с помощью полной репликации на уровне кластера и между ЦОДами. Для горизонтального масштабирования внутри кластера применяется вариант репликации с несколькими главными копиями (см. главу 6), называемый каузальной кластеризацией. Каузальная кластеризация поддерживает *каузальную согласованность*, гарантирующую, что причинно связанные операции видны всем клиентским приложениям в одном и том же порядке. Таким образом гарантируется, что клиентское приложение прочитает результат выполненных им же операций записи. Архитектура каузальной кластеризации показана на рис. 11.7 на примере кластера с тремя типами узлов: сервер приложений, основной сервер и сервер чтения. Серверы приложений исполняют код приложений и отправляют транзакции записи основным серверам, а запросы чтения – серверам чтения. Основные серверы асинхронно реплицируют все транзакции, применяя протокол Raft, который гарантирует долговечность, поскольку требует подтверждения записи от большинства серверов, прежде чем зафиксировать транзакцию. Основные серверы реплицируют транзакции серверам чтения, передавая журналы транзакций. Протокол Raft используется также для реализации различных архитектур репликации между ЦОДами для поддержки аварийного восстановления.

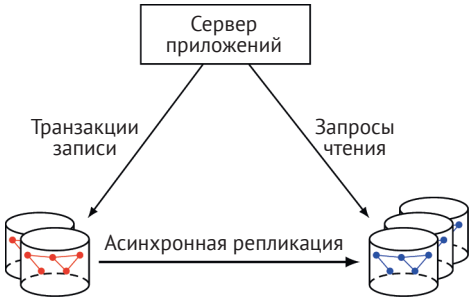


Рис. 11.7 ❖ Архитектура каузальной кластеризации Neo4j

В дополнение к высокой доступности каузальная кластеризация позволяет горизонтально масштабировать запросы к графу на несколько серверов чтения. Для оптимизации использования оперативной памяти Neo4j применяет секционирование с кешированием, при котором все запросы от одного и того же пользователя всегда передаются одному и тому же серверу чтения. Естественно, это повышает локальность ссылок на каждом сервере чтения и позволяет организовать масштабирование очень больших графов.

Из обсуждения выше следует, что максимальный размер графа ограничен емкостью диска основного сервера. Это ограничение позволяет линейно масштабировать производительность обхода пути, требуя в то же время хранить граф в максимально компактном виде. В Neo4j применяется динамическое сжатие указателей, чтобы расширять доступное адресное пространство по

мере необходимости и при этом находить соседние с узлом узлы и связи с помощью перехода по указателю. Наконец, дополнительные возможности для оптимизации открывает раздельное хранение узлов, связей и свойств. Так, в первых двух хранилищах можно хранить только основную информацию, так что записи узла и связи будут иметь фиксированный размер, а сложность обхода пути составит $O(1)$. В хранилище же свойств записи могут иметь переменный размер.

11.5.2. Другие графовые базы данных

Из других популярных графовых СУБД отметим Infinite Graph, Titan, GraphBase, Trinity и Sparksee.

11.6. Гибридные склады данных

Гибридные склады данных сочетают возможности различных складов данных и СУБД. Мы будем различать многомодельные NoSQL-системы и СУБД типа NewSQL.

11.6.1. Многомодельные NoSQL-системы

Цель многомодельных NoSQL-систем – уменьшить необходимость использования нескольких систем при разработке сложных приложений. Мы проиллюстрируем многомодельную NoSQL-систему на примере OrientDB – популярного склада данных типа NoSQL, объединяющего идеи объектно-ориентированной СУБД, документного хранилища NoSQL и графовой модели данных. Из других популярных многомодельных систем отметим ArangoDB и Microsoft Azure Cosmos DB.

История OrientDB началась с реализации на Java уровня хранения объектно-ориентированной СУБД для кластеров без разделения ресурсов (первоначально написанной на C++). Она предоставляет развитую модель данных со схемами, мощный язык запросов на основе SQL, оптимистические ACID-транзакции и поддержку высокой доступности и балансировки нагрузки.

Модель данных графовая с прямыми связями между записями. Существует четыре типа записей: Document, RecordBytes (двоичные данные), Vertex и Edge. Когда OrientDB создает запись (наименьшую единицу хранения), ей присваивается уникальный идентификатор Record ID.

Язык запросов представляет собой расширение SQL с добавлением обхода путей в графе. Поддерживаются различные виды индексов: SB-дерево (подразумевается по умолчанию), хеш-индекс для эффективного выполнения запросов с точным совпадением, полнотекстовый индекс Lucene для поиска в тексте и пространственный индекс Lucene для пространственных запросов.

Управление схемами навеяно объектно-ориентированными идеями и включает наследование классов. Класс определяет множество похожих

записей, он может быть бессхемным, с полной схемой (как в объектно-ориентированных базах данных) или с гибридной схемой. В режиме с гибридной схемой класс определяет атрибуты, но в некоторых записях могут быть дополнительные атрибуты. Наследование классов структурное, т. е. подкласс расширяет родительский класс, наследуя все его атрибуты.

Классы лежат в основе кластеризации и секционирования записей на несколько узлов. Каждый класс может иметь одну или несколько секций, называемых кластерами. При вставке новой записи в класс OrientDB выбирает, в какой кластер ее поместить, следуя одной из предопределенных стратегий:

- по умолчанию: выбирается кластер с идентификатором по умолчанию, определенным в классе;
- циклическая: кластеры классов связываются в кольцо, и новая запись помещается в следующий по порядку кластер;
- сбалансированная: проверяется количество записей в кластерах класса, и новая запись помещается в наименьший кластер;
- локальная: если база данных реплицирована, то выбирается главный кластер для текущего узла (тот, что обрабатывает вставки).

OrientDB поддерживает репликацию с несколькими главными копиями, т. е. все узлы кластера без разделения ресурсов могут параллельно записывать в базу данных. При обработке транзакции применяется оптимистическое многоверсионное управление конкурентностью, основанное на предположении о том, что конфликтов обновления немного. Поэтому транзакции обрабатываются без оглядки на обстановку до момента фиксации. В момент фиксации транзакции для каждой записи проверяется отсутствие конфликтов с другими транзакциями, и если таковые обнаружатся, то некоторые транзакции могут быть отменены.

11.6.2. СУБД типа NewSQL

NewSQL – недавно возникший класс СУБД, преследующий цель объединить масштабируемость NoSQL-систем с сильной согласованностью и удобством использования реляционных СУБД. Его основная задача – удовлетворить требования корпоративных информационных систем, в которых традиционно работали реляционные СУБД, но возникла потребность в масштабировании. NewSQL-системы обеспечивают масштабируемость, а также доступность, гибкие схемы и API для программирования информационно емких приложений. В предыдущих разделах мы видели, что обычно это достигается секционированием данных в кластерах без разделения ресурсов, состоящих из серийных компьютеров, и ослаблением требования согласованности базы данных. С другой стороны, реляционные СУБД обеспечивают сильную согласованность базы данных благодаря ACID-транзакциям и поддерживают стандартный язык SQL, чем облегчают работу инструментальным средствам и приложениям. Для них также имеются параллельные версии, допускающие масштабирование, но стоят они дорого, даже для кластера без разделения ресурсов.

Важный класс NewSQL-систем – гибридная транзакционно-аналитическая обработка (HTAP), цель которой – выполнение приложений OLAP и OLTP на

одних и тех же данных. HTAP-система позволяет проводить анализ оперативных данных в реальном масштабе времени и тем самым отказаться от традиционного разделения оперативной базы данных и хранилища данных и избежать сложностей, связанных с ETL.

NewSQL-системы появились недавно, и архитектуры у них разные. Однако можно выделить следующие общие черты: реляционная модель данных и стандартный SQL; ACID-транзакции; масштабируемость благодаря секционированию данных в кластере без разделения ресурсов; доступность благодаря репликации.

Далее в этом разделе мы проиллюстрируем NewSQL на примере систем Google F1 и LeanXcale. Из других NewSQL-систем отметим Apache Ignite, CockroachDB, Esgyn, GridGain, MemSQL, NuoDB, Splice Machine, VoltDB и SAP HANA.

11.6.2.1. F1

F1 – NewSQL-система производства компании Google, сочетающая масштабируемость Bigtable и согласованность и удобство работы, свойственные реляционным СУБД. Она была создана для приложения AdWords, крупномасштабного и выполняющего большое число операций обновления. F1 предлагает реляционную модель данных с некоторыми расширениями, полную поддержку SQL-запросов, с индексами и однократными запросами, и оптимистичные транзакции. Она построена поверх Spanner – масштабируемой системы хранения данных для кластеров без разделения ресурсов (см. раздел 5.5.1).

Модель данных в F1 реляционная с иерархической реализацией, идея которой заимствована у Bigtable. Несколько реляционных таблиц с зависимостями внешнего ключа можно организовать как вложенное отношение, так что строки каждой дочерней таблицы будут кластеризованы вместе со строками родительской таблицы на основе ключа соединения. Это позволяет ускорить обработку нескольких строк с одним и тем же внешним ключом и эффективно выполнять соединение. F1 также поддерживает табличные столбцы со структурированными типами данных, применяя Protocol Buffers – языково-независимый расширяемый механизм Google для сериализации структурированных данных. Механизм Protocol Buffers позволяет легко программировать преобразования между строками базы данных и структурами данных в памяти.

Основным интерфейсом является SQL, используемый как для OLTP-транзакций, так и для больших OLAP-запросов. Стандартный SQL расширен конструкциями для доступа к данным, хранящимся в буфере протокола. F1 также поддерживает соединение данных Spanner с данными из других источников, в частности Bigtable и CSV-файлов. F1 реализует интерфейс NoSQL-хранилищ ключей и значений с его быстрым доступом к строкам благодаря поддержке запросов с точным совпадением и по диапазону. Также поддерживается обновление по первичному ключу. Вторичные индексы хранятся в таблицах Spanner, ключом в них является конкатенация индексного ключа с первичным ключом индексируемой таблицы. Индексы в F1 могут быть

локальными или глобальными. Локальные индексы являются локальными относительно иерархии таблиц, в состав их индексных ключей входит первичный ключ корневой строки в качестве префикса. Записи индекса находятся в той же секции, что индексируемые ими строки, поэтому обновление индекса производится эффективно. С другой стороны, глобальные индексы охватывают несколько таблиц и не включают первичный ключ корневой строки в качестве префикса. Таким образом, они не могут находиться в одной секции с индексируемыми строками.

F1 поддерживает централизованное и распределенное выполнение запросов. Централизованно выполняются короткие OLTP-запросы, когда весь запрос обрабатывается в одном серверном узле F1. Распределенное выполнение используется для OLAP-запросов с высокой степенью параллелизма, для этого применяются методы потоковой обработки и повторного разбиения на основе хеширования.

В разделе 5.5.1 мы вкратце описали подход Spanner к горизонтальному масштабированию управления транзакциями. К свойствам Spanner следует также отнести отказоустойчивость, секционирование данных внутри ЦОДов, синхронную репликацию между территориально разнесенными ЦОДами и ACID-транзакции. В Spanner каждой транзакции назначается временная метка фиксации, которая используется для полного глобального упорядочения фиксаций. F1 поддерживает три типа транзакций, поверх поддержки транзакций, реализованной в Spanner:

- транзакции чтения с уровнем изоляции снимков с помощью временных меток снимков, имеющихся в Spanner;
- пессимистические транзакции с помощью ACID-транзакций на основе блокировок, имеющихся в Spanner;
- оптимистические транзакции с фазой чтения, в которой блокировки не нужны, и фазой проверки, в которой обнаруживаются конфликты на уровне строк (с помощью временных меток последней модификации строки) и принимается решение о фиксации или отмене.

11.6.2.2. LeanXcale

LeanXcale – NewSQL/HTAP-система с полной поддержкой SQL и полихранилища в кластере без разделения ресурсов. Она состоит из трех основных подсистем: хранения, запросов и транзакционной. Все они распределенные и отлично масштабируются (на сотни узлов).

LeanXcale предлагает полную функциональность SQL для реляционных таблиц с JSON-столбцами. Клиенты могут обращаться к LeanXcale из аналитических инструментов с помощью JDBC-драйвера. Важная особенность LeanXcale – доступ к полихранилищу с использованием скриптового языка запросов CloudMdsQL (см. раздел 11.7.3.2). Доступны разнообразные склады данных, в том числе исходные распределенные файлы данных (например, HDFS), параллельные базы данных SQL, базы данных NoSQL (например, MongoDB, запросы к которой можно записывать в виде программ на JavaScript).

В качестве подсистемы хранения используется проприетарное реляционное хранилище ключей и значений KiVi, допускающее эффективное гори-

зонтальное секционирование таблиц и индексов на основе первичного или индексного ключа. Каждая таблица хранится в таблице KiVi, ключ которой соответствует первичному ключу таблицы LeanXcale, а все столбцы хранятся в неизменном виде в столбцах KiVi. Индексы также хранятся в виде таблиц KiVi, причем индексные ключи отображаются на соответствующие первичные ключи. Эта модель обеспечивает высокую степень масштабируемости уровня хранения благодаря секционированию таблиц и индексов между узлами данных KiVi. KiVi предоставляет типичные для хранилищ ключей и значений операции put и get, а также все операции над одной таблицей: выборку по предикату, агрегирование, группировку и сортировку, т. е. все операторы реляционной алгебры, кроме соединения. Операции, включающие несколько таблиц, т. е. соединения, выполняются подсистемой запросов, и все алгебраические операторы выше соединения включаются в план запроса. Таким образом, все алгебраические операторы ниже соединения опускаются на уровень подсистемы хранения KiVi.

Подсистема запросов обрабатывает рабочие нагрузки OLAP, относящиеся к оперативным данным, так что аналитические запросы обрабатываются в режиме реального времени. Параллельная реализация подсистемы запросов устроена по принципу «одна программа, несколько потоков данных» (SPMD), в котором сочетаются межзапросный и внутриоператорный параллелизм. При этом несколько симметричных исполнителей (потоков) выполняют один и тот же запрос (оператор), но с разными участками данных.

В подсистеме запросов применена двухшаговая оптимизация запроса. После получения запроса план выполнения рассылается всем исполнителям и обрабатывается ими. Для параллельного выполнения добавлен шаг оптимизации, который преобразует сгенерированный последовательный план в параллельный. Преобразование включает замену последовательного просмотра таблиц параллельным и добавление операторов тасования, которые гарантируют, что в операторах, имеющих состояние (например, группировки или соединения), взаимосвязанные строки обрабатываются одним исполнителем. При параллельном просмотре строки базовых таблиц разделяются между всеми исполнителями, т. е. каждый исполнитель извлекает свое, не пересекающееся с другими подмножество строк. Для этого множество строк делится на части и полученные подмножества планируются различным экземплярам подсистемы запросов. Затем каждый исполнитель обрабатывает строки из подмножеств, запланированных его подсистеме запросов, обмениваясь строками с другими исполнителями, как то указано в операторах тасования, добавленных в план запроса. Для обработки соединений подсистема запросов поддерживает две стратегии обмена данными (тасование и широковещание) и различные методы соединения (хеширование, вложенные циклы и т. д.), выполняемые локально в каждом узле-исполнителе после завершения обмена данными.

Подсистема запросов спроектирована так, чтобы интегрировать данные из произвольных складов, в которых они хранятся в естественном формате и могут быть извлечены (параллельно) путем выполнения специальных скриптов или декларативных запросов. Это делает ее эффективным полихранилищем, которое способно обрабатывать данные прямо в исходном

формате, пользуясь всеми преимуществами выразительного скриптового языка и массового параллелизма. Кроме того, можно выполнять соединение наборов данных из разных источников, например HDFS или MongoDB, в т. ч. и таблиц LeanXcale, применяя эффективные параллельные алгоритмы соединения. Для поддержки однократных запросов к произвольному набору данных подсистема запросов умеет обрабатывать запросы на языке CloudMdsQL, обертывая скрипты платформенными подзапросами (раздел 11.7.3.2).

В разделе 5.5.2 мы познакомились с подходом LeanXcale к горизонтальному масштабированию управления транзакциями. Для этого LeanXcale подвергает свойства ACID декомпозиции и масштабирует каждое из них независимо, но так чтобы можно было осуществить композицию. Транзакционная подсистема обеспечивает сильную согласованность с помощью изоляции снимков. Операции чтения не блокируются операциями записи благодаря многоверсионному управлению конкурентностью. Поддерживается основанное на временных метках упорядочение и обнаружение конфликтов непосредственно перед фиксацией. Распределенный алгоритм поддержания транзакционной согласованности способен фиксировать транзакции полностью параллельно безо всякой координации, используя тщательно продуманное разделение обязанностей. Таким образом, видимость зафиксированных данных отделена от обработки фиксации. Поэтому обработку фиксации можно полностью распараллелить, не ставя под вопрос согласованность, которая регулируется видимостью зафиксированных обновлений. Все фиксации производятся параллельно, а когда образуется начальная группа зафиксированных транзакций без лакун, текущий снимок сдвигается вперед.

11.7. Полихранилища

Полихранилища предоставляют интегрированный доступ к нескольким облачным складам данных, будь то NoSQL, реляционная СУБД или HDFS. Обычно они поддерживают только запросы чтения, поскольку поддержка распределенных транзакций в гетерогенных складах данных – весьма трудная задача. Мы можем классифицировать полихранилища по уровню связанности с исходными складами данных: слабо связанные, сильно связанные и гибридные. В этом разделе мы приведем примеры систем из каждого класса, описав архитектуру и способы обработки запросов. И завершим раздел несколькими замечаниями.

11.7.1. Слабо связанные полихранилища

Слабо связанные полихранилища напоминают мультибазовые системы тем, что способны работать с автономными складами данных, к которым можно обращаться как через общий интерфейс полихранилища, так и через их собственные локальные API. Имеет место архитектура посредник–обертка с несколькими складами данных (например, NoSQL и реляционная СУБД),

изображенная на рис. 11.8. Каждый склад данных автономен, т. е. управляется локально и допускает доступ со стороны других приложений. Архитектура посредник–обертка, используемая в системах интеграции данных, может масштабироваться на большое число складов данных.

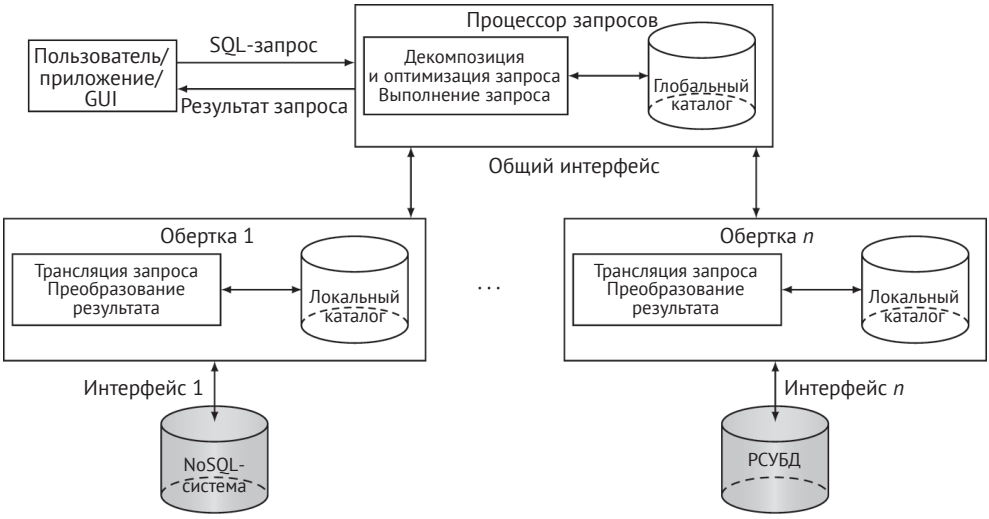


Рис. 11.8 ❖ Полихранилище со слабо связанной архитектурой

Основных модуля два: по одному процессору запросов и по одной обертке на каждый склад данных. Процессор запросов располагает каталогом складов данных, а у каждой обертки имеется локальный каталог ее склада. После того как все каталоги и обертки построены, процессор запросов может начать обработку входных запросов от пользователей, взаимодействуя с обертками. Обработка типичного запроса выглядит следующим образом:

1) проанализировать входной запрос и транслировать его в совокупность подзапросов (по одному на каждый склад данных), выраженных на общем языке, и интеграционного запроса;
2) отправить подзапросы соответствующим оберткам и преобразовать результаты в общий формат;
3) объединить результаты, полученные от оберток (для чего могут потребоваться операторы объединения и соединения), и вернуть итоговый результат пользователю. Ниже мы опишем три слабо связанных полихранилища: BigIntegrator, Forward и QoX.

11.7.1.1. BigIntegrator

BigIntegrator поддерживает SQL-подобные запросы и объединяет данные, хранящиеся в облачных складах данных на основе Bigtable, с данными в реляционных СУБД (необязательно облачных). Доступ к данным в Bigtable производится на языке Google Query Language (GQL) с очень ограниченными выразительными возможностями – например, в нем нет соединений,

а в команде выборки можно задавать только самые простые предикаты. Для компенсации ограничений GQL BigIntegrator предлагает механизм обработки запросов, основанный на плагинах, называемых поглотителем (absorber) и финализатором, которые дают возможность написать пред- и постобработку для операций, недоступных Bigtable. Например, предикат «LIKE» в команде выборки или соединение двух таблиц Bigtable будут обработаны процессором запросов BigIntegrator.

В BigIntegrator используется подход «локальная как представление» (ЛКП) (см. раздел 7.1.1) для определения глобальной схемы Bigtable и реляционных источников данных в виде плоских реляционных таблиц. Каждый источник может содержать несколько коллекций, представленных в виде исходной таблицы вида «имя-таблицы_имя-источника», где имя-таблицы – имя таблицы глобальной схемы, а имя-источника – имя источника данных. Например, имя «Employees_A» соответствует таблице Employees в источнике A, т. е. локальному представлению Employees. В SQL-запросах на исходные таблицы ссылаются как на таблицы.

На рис. 11.9 показана архитектура BigIntegrator с двумя источниками данных: реляционным и складом данных Bigtable. У каждой обертки имеется модуль импорта, а также плагины поглотителя и финализатора. Модуль импорта создает исходные таблицы и сохраняет их в локальном каталоге. Поглотитель выделяет из пользовательского запроса подзапрос (так называемый *фильтр доступа*), который выбирает данные из некоторой исходной таблицы, учитывая особенности источника. Обертка транслирует каждый фильтр доступа (созданный поглотителем) в оператор, называемый интерфейсной функцией, – свой для каждого вида источника. Интерфейсная функция используется для отправки запроса (в данном случае на языке GQL или SQL) источнику данных.

Обработка запроса состоит из трех шагов, в которых участвуют диспетчер поглотителей, оптимизатор запроса и диспетчер финализаторов. Диспетчер поглотителей принимает разобранный пользовательский запрос и для каждой упоминаемой в нем таблицы вызывает соответствующий поглотитель ее обертки. Чтобы заменить исходную таблицу фильтром доступа, поглотитель выделяет из запроса все исходные таблицы и, возможно, другие предикаты, учитывая возможности источника данных. Оптимизатор запросов переупорядочивает фильтры доступа и другие предикаты с целью сконструировать алгебраическое выражение, которое содержит обращения к фильтрам доступа и другим реляционным операторам. Он также выполняет стандартные преобразования, например проталкивание выборки вниз и привязку соединений. Диспетчер финализаторов принимает алгебраическое выражение и для каждого встречающегося в нем оператора доступа вызывает соответствующий финализатор из его обертки. Финализатор преобразует фильтры доступа в обращения к интерфейсным функциям.

Наконец, к исполнению запроса подключается процессор запросов, который интерпретирует алгебраическое выражение, вызывает интерфейсные функции для доступа к различным источникам данных и выполняет последующие реляционные операции в оперативной памяти.

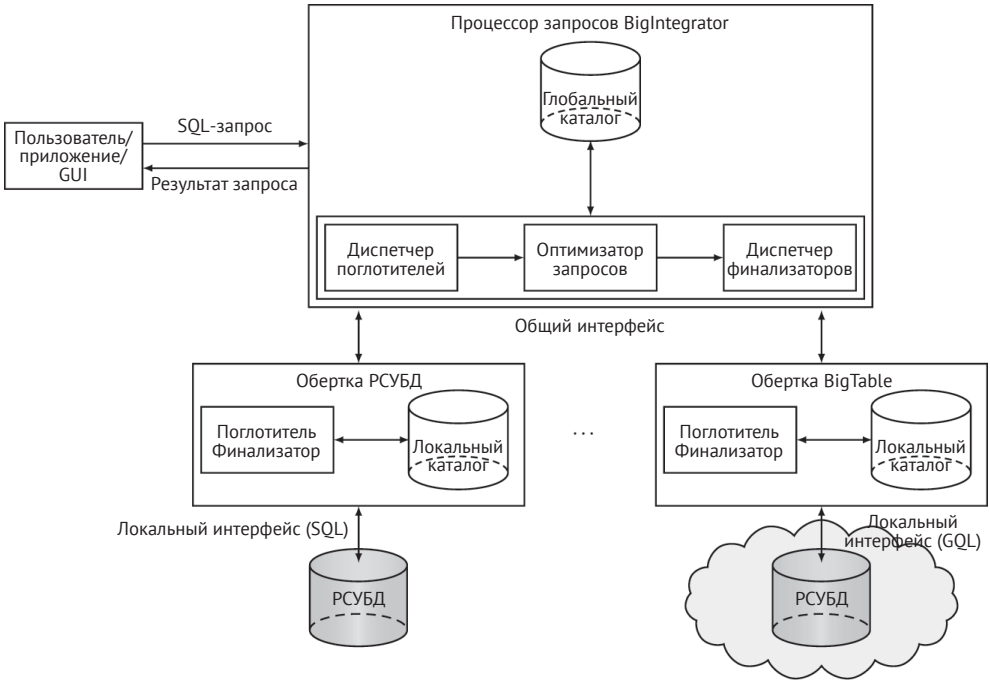


Рис. 11.9 ❖ Архитектура BigIntegrator

11.7.1.2. Forward

Система Forward поддерживает SQL++ – похожий на SQL язык, предназначенный для унификации моделей данных и языков запросов, применяемых в NoSQL и реляционных базах данных. В SQL++ имеется мощная слабо структурированная модель данных, расширяющая как JSON, так и реляционную модель. Forward также предоставляет развитую систему веб-разработки, в которой совместимость с JSON используется для интеграции с компонентами визуализации (например, Google Maps).

Дизайн SQL++ основан на том наблюдении, что основные понятия в обеих моделях похожи, например массив JSON аналогичен таблице SQL с упорядочением, а кортеж SQL – объектному литералу JSON. Таким образом, в SQL++ есть два вида коллекций: массив и «мешок», т. е. множество с дубликатами. Массив упорядочен (как массив JSON), и к каждому элементу можно обратиться по номеру позиции, а мешок не упорядочен (как таблица SQL). Кроме того, SQL++ расширяет реляционную модель, добавляя произвольную композицию сложных значений и гетерогенность элементов. Как во вложенных моделях данных, сложное значение может быть либо кортежем, либо коллекцией. Доступ к вложенным коллекциям производится с помощью вложенных выражений **`SELECT`** во фразе **`FROM`**. Можно также компоновать их с помощью оператора **`GROUP BY`**. Для сериализации вложенных коллекций служит оператор **`FLATTEN`**. В отличие от таблицы SQL, в которой все кортежи должны состоять из одинаковых атрибутов, в коллекции SQL++ допускаются

гетерогенные элементы, состоящие из смеси кортежей, скаляров и вложенных коллекций.

В Forward используется подход «глобальная как представление» (ГКП) (см. раздел 7.1.1), когда каждый источник данных (SQL или NoSQL) представляется пользователю виртуальным представлением SQL++, определенным над коллекциями SQL++. Таким образом, пользователь может предъявлять запросы на SQL++ с несколькими виртуальными представлениями. Архитектура Forward с процессором запросов и одной оберткой на каждый источник данных изображена на рис. 11.8. Процессор запросов выполняет декомпозицию запроса на SQL++, по максимуму используя возможности конкретного склада данных. Но, учитывая, что SQL++ не поддерживается напрямую источниками исходных данных, Forward разлагает запрос на один или несколько запросов на поддерживаемом языке и объединяет результаты, полученные в формате источника, стремясь компенсировать разрыв в семантике или выразительных возможностях между SQL++ и источником данных. Для стоимостной оптимизации запросов на SQL++ можно было бы позаимствовать из мультибазовых систем методы для работы с плоскими коллекциями. Но сделать это гораздо труднее из-за вложенности и гетерогенности элементов, присущих SQL++.

11.7.1.3. QoX

QoX – особый вид слабо связанного полихранилища, в котором запросами являются потоки аналитических управляемых данными работ (или потоки данных), которые интегрируют данные из реляционных СУБД и различных платформ, например MapReduce, или инструментов ETL. Типичный поток данных может объединять неструктурированные данные (например, твиты) со структурированными и включать как общие операции над данными – фильтрация, соединение, агрегирование, – так и определенные пользователем – анализ эмоциональной окраски или идентификацию товара. В системе применен новаторский подход к ETL, в котором на всех этапах процесса проектирования применяется набор показателей качества под названием QoX. Оптимизатор QoX принимает во внимание эти показатели и стремится оптимизировать выполнение потоков данных, так чтобы объединить тыловой ETL-конвейер и фронтальные операции запроса в единый аналитический конвейер.

Оптимизатор QoX использует xLM, проприетарный язык на основе XML, для представления потоков данных, которое, как правило, создается с помощью какого-то средства ETL. xLM описывает структуру потока: узлы соответствуют операциям и складам данных, а связывающие их ребра – таким важным свойствам этих операций, как тип операции, схема, статистика и параметры. С помощью оберток для трансляции xLM в диалект XML конкретного инструмента и обратно оптимизатор QoX может подключаться к внешним системам ETL и импортировать или экспортировать потоки данных.

Имея поток данных для нескольких складов данных и исполнительных систем, оптимизатор QoX оценивает стоимости альтернативных планов выполнения и порождает физический план (исполняемый код). Пространство

поиска, в котором ищутся эквивалентные планы выполнения, определяется преобразованиями потока данных, которые моделируют доставку данных (перемещение их туда, где будет выполняться операция), доставку функций (перемещение операции ближе к данным) и декомпозицию операций (на меньшие операции). Стоимость каждой операции оценивается на основе статистики (мощности, избирательности). Наконец, оптимизатор QoX порождает SQL-код для реляционных СУБД, код на языках Pig и Hive для MapReduce-систем и создает скрипты оболочки Unix для управления различными подпотоками, работающими в разных системах. Этот подход можно обобщить также на доступ к NoSQL-системам, если для них имеются SQL-подобные интерфейсы и обертки.

11.7.2. Сильно связанные полихранилища

Сильно связанные полихранилища ориентированы на эффективный опрос структурированных и неструктурированных данных для анализа (больших) данных. У них может быть и конкретная цель, например автонастройка или интеграция HDFS с реляционной СУБД. Однако все они жертвуют автономностью ради производительности, обычно в кластере без разделения ресурсов, так что к отдельным складам данных можно обратиться только через полихранилище.

Как слабо связанные системы, они предоставляют единый язык для запросов к структурированным и неструктурированным данным. Но процессор запросов напрямую использует локальные интерфейсы склада данных (см. рис. 11.10) или, в случае HDFS, может взаимодействовать с такой системой обработки данных, как MapReduce или Spark. Таким образом, процессор запросов получает прямой доступ к складам данных, что позволяет эффективно перемещать данные между складами. Однако количество складов данных, с которыми реализован интерфейс, обычно очень ограничено.

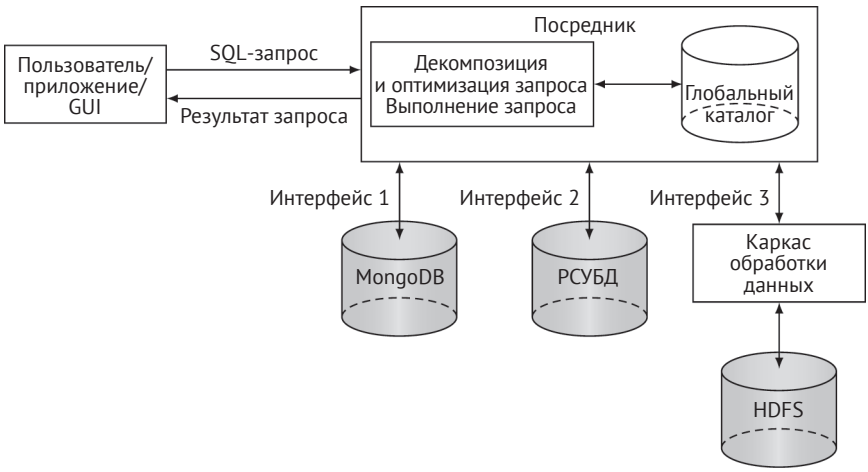


Рис. 11.10 ❖ Полихранилище с сильно связанной архитектурой

Далее в этом разделе мы опишем три репрезентативных сильно связанных хранилища: Polybase, HadoopDB и Estocada. Есть еще три интересные системы: Redshift Spectrum, Odyssey и JEN. Amazon Redshift Spectrum – одна из функций хранилища данных Amazon Redshift на облачной платформе Amazon Web Services (AWS). Она позволяет выполнять SQL-запросы к большим неструктурированным данным, хранящимся под управлением службы Amazon Simple Storage Service (S3). Полихранилище Odyssey может работать с различными аналитическими системами, например параллельной OLAP-системой или Hadoop. Оно позволяет хранить и опрашивать данные, хранящиеся в HDFS и реляционных СУБД, используя ситуативно материализуемые представления (opportunistic materialized view) и метод MISO для настройки физической структуры полихранилища (Hive/HDFS или реляционные СУБД), т. е. принятия решения о том, где лучше хранить данные, чтобы повысить производительность обработки больших данных. Промежуточные результаты выполнения запроса рассматриваются как ситуативно материализуемые представления, которые затем можно поместить в базовое хранилище, чтобы оптимизировать выполнение последующих запросов. JEN – это компонент, надстроенный над HDFS с целью обеспечения тесной связи с реляционной СУБД. Он позволяет соединять данные из обоих складов, применяя параллельные алгоритмы соединения, в частности эффективный алгоритм зигзагообразного соединения, а также методы, позволяющие минимизировать перемещение данных. По мере увеличения размера данных выполнение соединения на стороне HDFS становится более эффективным.

11.7.2.1. Polybase

Polybase – одна из функций Microsoft SQL Server Parallel Data Warehouse (PDW), которая позволяет пользователю опрашивать неструктурированные (HDFS) данные, хранящиеся в кластере Hadoop, с помощью SQL и интегрировать их с реляционными данными, хранящимися в PDW. HDFS-файл в кластере Hadoop можно рассматривать как внешнюю таблицу и использовать его наряду с собственными таблицами PDW в SQL-запросах. Polybase пользуется всеми возможностями PDW, параллельной СУБД в кластере без разделения ресурсов. Благодаря оптимизатору запросов PDW SQL-операторы, воздействующие на HDFS-данные, транслируются в задания MapReduce, которые исполняются прямо в кластере Hadoop. Кроме того, HDFS-данные можно параллельно экспортировать в PDW и импортировать из PDW, используя ту же службу PDW, которая отвечает за раздачу данных вычислительным узлам (тасование).

Архитектура полихранилища Polybase, интегрированного в PDW, показана на рис. 11.11. Polybase пользуется службой перемещения данных (Data Movement Service – DMS) PDW, которая занимается раздачей промежуточных данных узлам PDW, например для того, чтобы кортежи, соответствующие друг другу в операции эквисоединения, находились в том узле, который эту операцию выполняет. DMS дополнена компонентом HDFS Bridge, который отвечает за все коммуникации с HDFS. HDFS Bridge также позволяет экземплярам DMS параллельно обмениваться данными с HDFS (напрямую обращаясь к секциям HDFS).

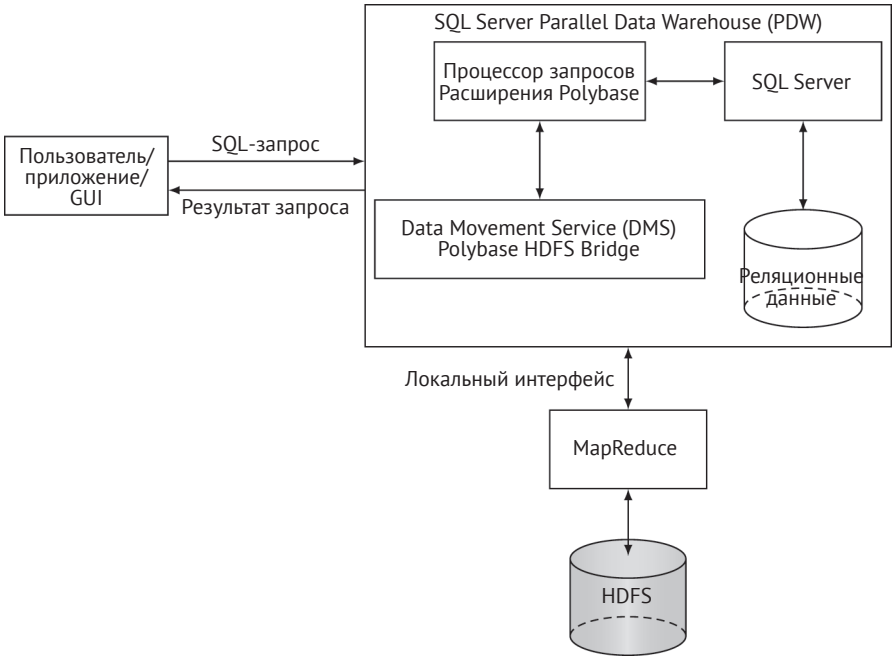


Рис. 11.11 ❖ Архитектура Polybase

Polybase опирается на стоимостной оптимизатор запросов PDW для решения вопроса о том, когда выгодно перенести операции SQL над HDFS-данными в кластер Hadoop для выполнения. Поэтому нужна детальная статистика внешних таблиц, которую можно получить путем исследования статистически значимой выборки из таблиц HDFS. Оптимизатор запросов перебирает эквивалентные ПВС и выбирает план с наименьшей стоимостью. Пространство поиска образовано множеством декомпозиций запроса на две части: одна выполняется как задания MapReduce в кластере Hadoop, другая – как регулярные реляционные операторы на стороне PDW. Задания MapReduce можно использовать для операций выборки и проекции с внешними таблицами, а также для соединения двух внешних таблиц. Затем результаты выполнения заданий можно экспортировать в PDW и там соединить с реляционными данными, применив параллельные алгоритмы хеш-соединения.

Важное ограничение при выполнении операций над HDFS-данными с помощью заданий MapReduce заключается в том, что даже для простых операций поиска задержка велика. Для решения проблемы в Polybase используется индекс над внешними HDFS-данными, имеющий структуру B+-дерева и хранящийся в PDW. При этом применяется надежный и эффективный код индексирования, имеющийся в PDW, и не требуется хранить или кешировать внутри PDW все HDFS-данные, что привело бы к непомерному увеличению размера. Таким образом, оптимизатор может использовать индекс как предварительный фильтр, чтобы сократить объем работы, выполняемой в виде заданий MapReduce. Для синхронизации индекса с данными, хранящимися в HDFS, применяется инкрементный подход – регистрируется, что индекс

устарел, а затем он лениво перестраивается. Для обработки запроса, поступившего во время переиндексирования, одна его часть выполняется с применением индекса в PDW, а другая – в виде задания MapReduce, но только над данными в HDFS, которые изменились. В Apache AsterixDB применяется похожий подход к индексированию внешних данных, хранящихся в HDFS, что позволяет предъявлять запросы как к данным, управляемым AsterixDB, так и к внешним данным в HDFS.

11.7.2.2. HadoopDB

Цель HadoopDB – взять лучшее из параллельной СУБД (высокопроизводительный анализ структурированных данных) и систем на основе MapReduce (масштабируемость, отказоустойчивость и гибкость обработки неструктурированных данных), используя при этом SQL-подобный язык (HiveQL) и реляционную модель данных. Для этого HadoopDB тесно связывает каркас Hadoop, включая MapReduce и HDFS, с нераспределенной реляционной СУБД, развернутой в кластере без разделения ресурсов, как в случае параллельной СУБД.

HadoopDB дополняет архитектуру Hadoop четырьмя компонентами: соединитель с базой данных, каталог, загрузчик данных и планировщик SQL-MapReduce-SQL (SMS). Соединитель с базой данных предоставляет обертки реляционной СУБД с помощью JDBC-драйверов. В каталоге, который представлен XML-файлом в HDFS, хранится информация о базах данных, которая используется при обработке запросов. Загрузчик данных отвечает за первичное и повторное секционирование коллекций (ключ, значение) с помощью хеширования по ключу и за загрузку данных в базы с несколькими секциями. Планировщик SMS расширяет Hive, компонент Hadoop, преобразующий HiveQL в задания MapReduce, которые подключаются к таблицам, хранящимся в виде файлов в HDFS. Эта архитектура приводит к экономически эффективной параллельной реляционной СУБД, в которой данные секционированы как на уровне таблиц реляционной СУБД, так и на уровне HDFS-файлов, причем секции могут находиться в одном и том же узле кластера для удобства параллельной обработки.

Обработка запросов устроена просто: планировщик SMS занимается трансляцией и оптимизацией, а MapReduce – выполнением. Задача оптимизации – передать как можно больше работы базам данных с одним узлом и повторно секционировать коллекции данных по мере необходимости. Планировщик SMS разлагает HiveQL-запрос в ПВЗ, состоящий из реляционных операторов. Затем операторы транслируются в задания MapReduce, а листовые узлы плана снова преобразуются в SQL для опроса экземпляров реляционных СУБД. В MapReduce этапу редукции должно предшествовать повторное секционирование. Но если оптимизатор обнаруживает, что таблица секционирована по столбцу, используемому как ключ агрегирования в операции Reduce, то он упрощает ПВЗ, преобразуя его в задание, содержащее только операцию Map и оставляя агрегирование узлам реляционной СУБД. Аналогично повторного секционирования можно избежать для эквисоединений, если обе части секционированы по ключу соединения.

11.7.2.3. Estocada

Estocada – самонастраиваемое полихранилище, цель которого – оптимизировать производительность приложений, которые должны работать с данными в нескольких моделях, в т. ч. реляционных, ключ-значение, документных и графовых. Чтобы добиться максимально возможной производительности от всех доступных складов данных, Estocada автоматически распределяет и секционирует данные между складами, которые полностью контролируются им и, следовательно, не являются автономными. Таким образом, это сильно связанное полихранилище.

Распределение данных производится динамически и опирается на комбинацию эвристических и основанных на стоимости решений, принимающих во внимание паттерны доступа к данным (после того как они проявляются). Каждая коллекция данных хранится в виде набора секций, содержимое которых может перекрываться, а каждая секция может храниться на любом из имеющихся складов данных. Таким образом, может случиться, что секция хранится на складе, модель данных которых отличается от «родной». Чтобы сделать приложения Estocada независимыми от склада данных, каждая секция внутри описывается как материализованное представление одной или нескольких коллекций данных. Таким образом, обработка запроса включает его переписывание на основе представлений.

Estocada поддерживает два вида запросов – для хранения и опроса данных – и состоит из четырех основных модулей: консультант по хранению, каталог, процессор запросов и движок выполнения. Эти компоненты могут напрямую обращаться к складам данных через локальные интерфейсы последних. Процессор запросов имеет дело только с запросами, выраженными в терминах одной модели на языке запросов соответствующего источника данных. Однако для интеграции различных источников необходимо иметь общие модель данных и язык, надстроенные над Estocada. Консультант по хранению отвечает за секционирование коллекций данных и делегирование хранения секций различным складам. Для самонастройки приложений он также может рекомендовать повторное секционирование или перемещение данных с одного склада на другой, принимая во внимание паттерны доступа. Каждая секция определена как материализованное представление, выраженное в виде запроса к коллекции на «родном» для нее языке. В каталоге хранится информация о секциях, в т. ч. о стоимости операций доступа; стоимость привязывается к паттернам, специфичным для конкретных складов.

Пользуясь каталогом, процессор запросов преобразует запрос к коллекции данных в логический ПВС, возможно, включающий несколько складов данных (если секции коллекции хранятся на разных складах). Для этого исходный запрос переписывается в терминах материализованных представлений, ассоциированных с коллекцией данных, и выбирается вариант, наилучший с точки зрения оценочной стоимости выполнения. Движок выполнения транслирует логический ПВС в физический, который можно выполнить непосредственно, разделив работу между складами данных и исполняющей средой выполнения Estocada, которая предоставляет собственные операторы (выборки, соединения, агрегирования и т. д.).

11.7.3. Гибридные системы

Гибридные системы стремятся соединить преимущества слабо связанных систем (например, доступ ко многим разным складам данных) и сильно связанных систем (например, эффективный доступ к некоторым складам через их локальные интерфейсы). Поэтому применяется общая архитектура посредник–обертка (см. рис. 11.12), но процессор запросов может напрямую обращаться к некоторым складам данных, например к HDFS посредством MapReduce или Spark.

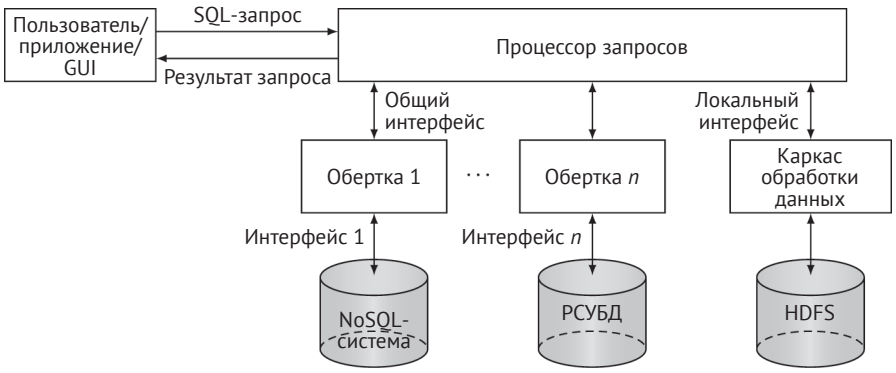


Рис. 11.12 ❖ Архитектура гибридного полихранилища

Мы опишем три гибридных полихранилища: Spark SQL, CloudMdsQL и BigDAWG.

11.7.3.1. Spark SQL

Spark SQL – это модуль системы Apache Spark, предназначенный для интеграции обработки реляционных данных с API функционального программирования Spark. Он поддерживает SQL-подобные запросы, которые способны интегрировать данные HDFS, доступные через Spark, с внешними источниками данных (например, реляционными базами), доступными через обертки. Таким образом, это гибридное полихранилище с сильной связью Spark/HDFS и слабой связью с внешними источниками данных.

В Spark SQL принята вложенная реляционная модель данных. Он поддерживает все основные типы данных SQL, а также пользовательские и составные типы данных (структуры, массивы, отображения и объединения), которые допускают произвольную вложенность. Поддерживается также тип данных DataFrame – распределенная коллекция строк с общей схемой, напоминающая реляционную таблицу. Объект DataFrame можно сконструировать из таблицы во внешнем источнике данных или из существующего гибкого распределенного набора данных (RDD) Spark, состоящего из объектов Java или Python. Построенными объектами DataFrame можно манипулировать

в процедурном коде Spark с помощью различных реляционных операторов, в частности WHERE и GROUPBY, которые принимают выражения.

На рис. 11.13 показана архитектура Spark SQL, работающего в виде библиотеки поверх Spark. Процессор запросов обращается к Spark напрямую через интерфейс Spark с Java, а к внешним источникам данных (например, к реляционной СУБД или к хранилищу ключей и значений) через общий интерфейс Spark SQL, поддерживаемый обертками (драйверами JDBC). Процессор запросов состоит из двух основных компонент: DataFrame API и оптимизатора запросов Catalyst. DataFrame API обеспечивает тесную интеграцию между реляционной и процедурной обработками, позволяя выполнять реляционные операции с внешними источниками данных и RDD. Он интегрирован с поддерживаемыми Spark языками программирования (Java, Scala, Python) и допускает простое встраиваемое определение пользовательских функций без сложного процесса регистрации, типичного для других систем баз данных. Таким образом, DataFrame API позволяет разработчикам органично сочетать реляционное и процедурное программирование, например выполнять сложный (с трудом выражаемый на SQL) анализ больших коллекций данных (доступных с помощью реляционных операций).

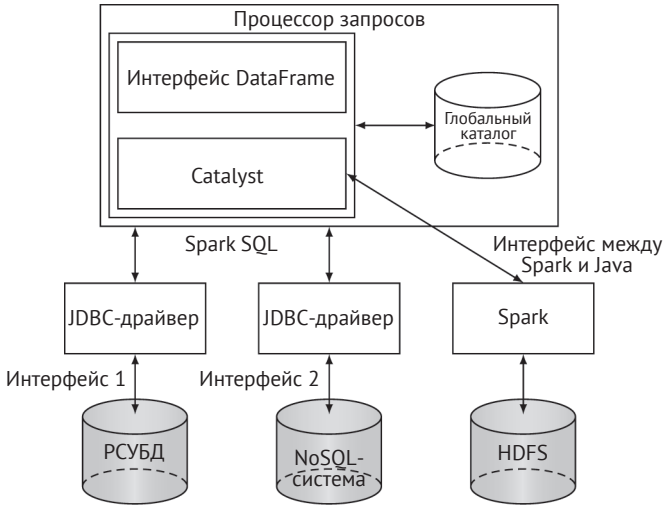


Рис. 11.13 ❖ Архитектура Spark SQL

Catalyst представляет собой расширяемый оптимизатор запросов, поддерживающий как стоимостную оптимизацию, так и оптимизацию на основе правил. За расширяемым дизайном стоит желание упростить добавление новых методов оптимизации, например поддержать новые возможности Spark SQL, а также дать разработчикам возможность расширять оптимизатор с целью поддержки внешних источников данных, например путем добавления зависящих от источника правил перемещения вниз предикатов выборки. Хотя расширяемые оптимизаторы запросов предлагались и раньше, обычно

они требовали сложного языка задания правил и специального компилятора для трансляции правил в исполняемый код. Catalyst же использует стандартные возможности языка функционального программирования Scala, в частности сопоставление с образцами, чтобы разработчикам было проще задавать правила, компилируемые в байт-код Java.

Catalyst предлагает общий каркас преобразований для представления дерева запроса и применения к нему правил манипулирования. Этот каркас используется на четырех этапах: (1) анализ запроса, (2) логическая оптимизация, (3) физическая оптимизация, (4) генерация кода. На этапе анализа запроса имена разрешаются с помощью каталога (содержащего информацию о схеме) и порождается логический план. На этапе логической оптимизации к логическому плану применяются стандартные оптимизации, основанные на правилах, например: опускание предикатов, распространение null и упрощение булевых выражений. На этапе физической оптимизации перебираются эквивалентные физические планы, получаемые подстановкой физических операторов, реализованных в движке выполнения Spark или во внешних источниках данных. Из этих планов выбирается оптимальный, для чего используется простая модель стоимости, в частности для сравнения алгоритмов соединения. Генерация кода опирается на язык Scala, в том числе на имеющиеся в нем простые средства построения абстрактных синтаксических деревьев (АСД). Затем АСД можно подать на вход компилятора Scala на этапе выполнения и сгенерировать байт-код Java, который исполняется вычислительными узлами.

Для ускорения выполнения запросов в Spark SQL применяется кеширование в оперативной памяти часто используемых данных, для чего используется столбцовое хранилище (т. е. коллекции данных хранятся по столбцам, а не по строкам). По сравнению со встроенным в Spark кешем, в котором данные хранятся как Java-объекты, такой столбцовый кеш может уменьшить объем потребляемой памяти на порядок благодаря использованию схем сжатия (например, кодирование по словарю и кодирование длин серий). Кеширование особенно полезно для обработки интерактивных запросов и в итеративных алгоритмах машинного обучения.

11.7.3.2. CloudMdsQL

CloudMdsQL поддерживает мощный функциональный SQL-подобный язык, спроектированный для опроса нескольких гетерогенных источников данных (например, реляционных и NoSQL). Запрос на CloudMdsQL может содержать вложенные подзапросы, каждый подзапрос напрямую адресует конкретный склад данных и может содержать внедренные обращения к собственным интерфейсам запросов этого склада. Таким образом, главное новшество заключается в том, что запрос на CloudMdsQL может задействовать всю мощь локальных складов данных, разрешая вызывать их собственные средства запросов (например, поиск в ширину в графовой базе данных) как функции. CloudMdsQL был расширен на системы распределенной обработки, например Apache Spark, за счет применения определенных пользователями ситуативных операторов отображения, фильтрации и редукции в качестве подзапросов.

Язык CloudMdsQL основан на SQL и дополнен средствами внедрения подзапросов, выраженных в терминах собственного интерфейса запросов каждого склада данных. В основе общей модели данных лежат таблицы. Поддерживаются развитые типы данных, на которые можно отобразить типы базовых складов данных, например массивы и JSON-объекты, что необходимо для обработки составных и вложенных данных; имеются также основные операторы, действующие на такие составные типы. CloudMdsQL позволяет определять именованные табличные выражения как функции на Python, это полезно для опроса складов данных, имеющих только интерфейс запросов, основанный на API. CloudMdsQL-запрос выполняется в контексте однократной схемы, образованной всеми именованными табличными выражениями, встречающимися в запросе. Такой подход компенсирует отсутствие глобальной схемы и позволяет компилятору запросов выполнять семантический анализ запроса.

Дизайн языка CloudMdsQL отражает тот факт, что система работает в облаке, где места установки ее компонент строго контролируются. Архитектура движка обработки запросов полностью распределенная, узлы могут напрямую взаимодействовать между собой, обмениваясь кодом (планами запросов) и данными. Такая архитектура открывает интересные возможности для оптимизации, например можно минимизировать объем передачи данных, пересылая наименьшее количество промежуточных данных для дальнейшей обработки одним узлом. Каждый узел движка обработки запросов состоит из двух частей (мастер и исполнитель) и размещен в каждом узле компьютерного кластера, где имеется склад данных. У каждого мастера и исполнителя имеется коммуникационный процессор, который поддерживает операции отправки и получения для обмена данными и командами между узлами. Мастер принимает на входе запрос и, пользуясь планировщиком запросов и каталогом (где хранятся метаданные и информация о стоимости для всех источников данных), порождает план запроса, а затем отправляет его одному узлу движка обработки запросов для выполнения. Каждый исполнитель играет роль облегченной исполняющей среды, надстроенной над складом данных, и состоит из трех общих модулей (разделяющих один и тот же библиотечный код) – контроллера выполнения запроса, подсистемы операторов и хранилища таблиц – одного модуля-обертки, специфичного для конкретного склада данных.

Планировщик запросов выполняет стоимостную оптимизацию. Для сравнения вариантов переписывания запроса оптимизатор пользуется простым каталогом, содержащим базовую информацию о коллекциях на складе данных – мощности, избирательности атрибутов, индексы, – а также простой моделью стоимости. Информацию можно раскрывать с помощью оберток в форме функций стоимости или статистики базы данных. Язык запросов также позволяет пользователю самостоятельно задать функции стоимости и избирательности, если их нельзя получить из каталога, как правило, в случае использования подзапросов на «родном» языке склада данных. Пространство поиска планов образовано традиционными преобразованиями, например: перемещение вниз предикатов выборки, использование соединения с привязкой (bind join), определение порядка соединений или планирование доставки промежуточных данных.

11.7.3.3. BigDAWG

Как и мультибазовые системы, все рассмотренные до сих пор полихранилища обеспечивают прозрачный доступ к нескольким складам данных, предлагая общую модель данных и язык. Система BigDAWG (Big Data Analytics Working Group) пошла по другому пути, ее цель – унифицировать средства запросов в условиях различных моделей данных и языков. Таким образом, ни общей модели данных, ни единого языка в ней не существует. Основная абстракция в BigDAWG – информационный остров, т. е. набор складов данных, для доступа к которым используется один и тот же язык запросов. Таких островов может быть несколько: реляционные СУБД, массивные СУБД (array DBMS), NoSQL и системы потоков данных (СПД). Внутри острова склады данных слабо связаны, т. е. необходима обертка (называемая *прокладкой*, англ. shim), которая отображает язык острова на «родной» язык склада. Если запрос обращается к нескольким складам данных, то, возможно, придется копировать объекты из одной локальной базы данных в другую, воспользовавшись операцией **`CAST`**, которая предоставляет форму сильной связи. Именно поэтому BigDAWG можно считать гибридным полихранилищем.

Архитектура BigDAWG в высокой степени распределенная, в ней имеется тонкий слой для интерфейса между информационными островами и инструментами (например, визуализации) и приложениями. Поскольку общей модели и языка не существует, нет и единого процессора запросов. Вместо этого у каждого информационного острова имеется свой процессор запросов. Обработка запроса внутри острова производится примерно так же, как в мультибазовых системах: большая ее часть передается складам данных, а процессор запросов только объединяет результаты. Оптимизатор запросов не пользуется моделью стоимости, но принимает во внимание эвристики и знание о высокой производительности некоторых складов. Для простых запросов, например выборка–проекция–соединение, оптимизатор использует доставку функций, чтобы минимизировать перемещение данных и объем сетевого трафика. Для сложных запросов, например аналитических, оптимизатор может рассмотреть в качестве варианта доставку данных, т. е. перемещение данных на склад, способный обеспечить наивысшую производительность.

Запрос, адресованный некоторому острову, может затрагивать несколько островов. В таком случае запрос необходимо выразить в виде нескольких подзапросов, каждый на языке одного острова. Чтобы указать остров, которому предназначен подзапрос, пользователь помещает подзапрос внутри спецификации `SCOPE` (область действия). Таким образом, в многоостровном запросе будет несколько областей действия, описывающих ожидаемое поведение подзапросов. Кроме того, пользователь может вставить операции **`CAST`** для эффективного перемещения промежуточных наборов данных между островами. Следовательно, обработка многоостровных запросов определяется тем, как пользователь задаст подзапросы, области действия и операции **`CAST`**.

11.7.4. Заключительные замечания

Все полихранилища имеют общую цель – опрос нескольких складов данных, но к этой цели ведет много путей, зависящих от функциональных требова-

ний. И вот эти-то требования оказывают важное влияние на принятие проектных решений. Доминирует тенденция к интеграции реляционных данных (хранящихся в реляционных СУБД) с данными на других складах, например HDFS (Polybase, HadoopDB, Spark SQL, JEN) или NoSQL (Bigtable – для BigIntegrator, документные хранилища – для Forward). Стало быть, важное различие – виды поддерживаемых складов данных. Например, Estocada, BigDAWG и CloudMdsQL могут поддержать широкий круг складов данных, тогда как Polybase и JEN нацелены только на интеграцию реляционных СУБД с HDFS. Отметим также растущую важность доступа к файловой системе HDFS, используемой в Hadoop, в частности при работе со Spark, – это основной сценарий интеграции структурированных и слабо структурированных данных.

Еще одна тенденция – появление самонастраивающихся полихранилищ, таких как Estocada и Odyssey, цель которых – задействовать имеющиеся склады данных для повышения производительности. Что касается модели данных и языка запросов, то большинство систем предоставляет абстракцию по образцу SQL. Однако в QoX применяется более общая абстракция графов для описания аналитических потоков данных. А Estocada и BigDAWG допускают обращение к складам данных на их собственных языках (или на языке острова). CloudMdsQL также допускает платформенные запросы, но в виде подзапросов на SQL-подобном языке.

Многие полихранилища предлагают ту или иную поддержку управления глобальной схемой, применяя подходы ГКП или ЛКП с некоторыми вариациями, например в BigDAWG используется ГКП внутри информационных островов (с общей моделью данных). Однако QoX, Estocada, Spark SQL и CloudMdsQL не поддерживают глобальные схемы, хотя и предлагают средства для работы с локальными схемами складов данных.

Для обработки запросов применяются методы, заимствованные из распределенных систем баз данных, например доставка данных или функций, декомпозиция запроса (с учетом возможностей складов данных и с применением соединения с привязкой, опускания выборки и т. п.). Оптимизация запросов обычно также поддерживается – с помощью простой модели стоимости или эвристик.

11.8. Заключение

По сравнению с традиционными реляционными СУБД новые технологии обещают улучшить масштабируемость, производительность и упростить использование. Они также дополняют новые технологии управления большими данными (см. главу 10).

Основная мотивация NoSQL – устранить основные ограничения реляционных СУБД: универсальный подход ко всем видам данных и приложений и ограниченную масштабируемость и доступность базы данных в облаке, а также найти компромисс между строгой согласованностью и доступностью базы данных, которые, согласно теореме CAP, одновременно недостижимы. Выделяют четыре основные категории NoSQL-систем в зависимости от базовой модели данных: хранилища ключей и значений, хранилища

с широкими столбцами, документные хранилища и графовые базы данных. Мы проиллюстрировали каждую категорию репрезентативным примером системы: DynamoDB (хранилище ключей и значений), Bigtable (хранилище с широкими столбцами column), MongoDB (документное хранилище) и Neo4j (графовая база данных). Мы также остановились на многомодельной NoSQL-системе OrientDB, которая сочетает идеи объектно-ориентированной, документной и графовой моделей данных.

NoSQL-системы предлагают масштабируемость, а также доступность, гибкие схемы и практичный API, но в общем случае это достигается ценой ослабления сильной согласованности базы данных. NewSQL – недавно появившийся класс СУБД, призванный объединить масштабируемость NoSQL-систем с сильной согласованностью и удобством реляционных СУБД. Цель состоит в том, чтобы удовлетворить требования корпоративных информационных систем, традиционно обслуживаемых реляционными СУБД, которые, однако, нуждаются в возможности масштабирования. Мы проиллюстрировали NewSQL на примере СУБД Google F1 и LeanXcale.

Для построения облачных информационно емких приложений зачастую необходимо использовать несколько складов данных (NoSQL, HDFS, реляционные СУБД, NewSQL), каждый из которых оптимизирован под конкретный вид данных и задач. В частности, во многих случаях требуется комбинировать слабо структурированные данные (например, файлы журналов, твиты, веб-страницы), которые лучше поддерживают HDFS или NoSQL-системы, с более структурированными данными, хранящимися в реляционных СУБД. Полихранилища предоставляют интегрированный или прозрачный доступ к различным облачным складам данных с помощью одного или нескольких языков запросов. Мы классифицировали полихранилища по уровню связанности обслуживаемых складов данных: слабо связанные, сильно связанные и гибридные. Затем мы представили репрезентативные полихранилища для всех трех классов: BigIntegrator, Forward и QoX (слабо связанные); Polybase, HadoopDB и Estocada (сильно связанные); Spark SQL, CloudMdsQL и BigDAWG (гибридные).

В настоящее время доминирует тенденция к интеграции реляционных данных (хранящихся в реляционных СУБД) с другими видами складов данных, например HDFS или NoSQL. Но между полихранилищами имеются существенные различия в плане поддерживаемых складов. Мы также отметили растущую важность доступа к файловой системе HDFS, особенно в каркасах для обработки больших данных типа MapReduce или Spark. Еще одна тенденция – появление самонастраивающихся полихранилищ, цель которых – задействовать имеющиеся склады данных для повышения производительности. Что касается модели данных и языка запросов, то большинство систем предоставляет абстракцию по образцу реляционной модели и SQL. Однако в QoX применяется более общая абстракция графов для описания аналитических потоков данных. А Estocada и BigDAWG допускают обращение к складам данных на их собственных языках. Для обработки запросов применяются методы, заимствованные из распределенных систем баз данных (см. главу 4) и получившие дальнейшее развитие.

11.9. Библиографические замечания

Ландшафт в области NoSQL, NewSQL и полихранилищ продолжает изменяться, а стандартов пока нет. Поэтому составить хорошую актуальную библиографию трудно. На эту тему существует много книг и научных статей, но все они быстро устаревают. Более-менее актуальную информацию можно найти на сайтах соответствующих систем и в блогах. В этой главе мы сконцентрировали внимание на принципах и архитектуре систем, а не на деталях реализации, которые со временем меняются.

Часто можно встретить утверждение, что побудительной причиной для NoSQL является теорема CAP, позволяющая понять, в чем состоит компромисс между согласованностью (C), доступностью (A) и устойчивостью к разделению сети (P). Первоначально она была высказана в виде гипотезы [Brewer 2000], а затем доказана в работе [Gilbert and Lynch 2002]. Хотя теорема CAP ничего не говорит о масштабируемости, в некоторых NoSQL-системах ее цитируют в качестве оправдания для отсутствия поддержки ACID-транзакций.

Есть несколько книг, содержащих введение в NoSQL, в частности [Strauch 2011, Redmond and Wilson 2012], в которых предмет хорошо освещается на примере нескольких репрезентативных систем, описанных в этой главе. Имеются также хорошие книги по конкретным системам. Описание документного хранилища MongoDB основано на книге [Plugge et al. 2010], а также информации с сайта MongoDB. AsterixDB [Alsubaiee et al. 2014] и Couchbase [Borkar et al. 2016] – хранилища документов в формате JSON, поддерживающие диалект SQL++, первоначально предложенный в работе [Ong et al. 2014]. Имеется также отличная книга по SQL++, написанная для практикующих программистов [Chamberlin 2018] Доном Чемберлином, одним из авторов оригинального языка SQL. Механизм доступа к внешним данным AsterixDB и их индексирования описан в работе [Alamoudi et al. 2015]. Существует также хорошее описание DynamoDB [DeCandia et al. 2007] и Bigtable [Chang et al. 2008]. По графовым базам данных и Neo4j есть великолепная книга от авторов Neo4j [Robinson et al. 2015]. В Neo4j используется каузальная модель согласованности [Elbushra and Lindström 2015] для репликации с несколькими главными копиями и протокол Raft обеспечения долговечности транзакций [Ongaro and Ousterhout 2014].

Раздел, посвященный NewSQL-системам, основан на описаниях СУБД F1 [Shute et al. 2013] и LeanXcale [Jimenez-Peris and Patiño Martinez 2011, Kolev et al. 2018].

Хорошее изложение побудительных мотивов для создания полихранилищ имеется в работах [Duggan et al. 2015, Kolev et al. 2016b]. Раздел о полихранилищах основан на нашей обзорной статье по обработке запросов в системах с несколькими складами [Bondiombouy and Valduriez 2016]. В этой статье выделено три класса систем: (1) слабо связанные, (2) сильно связанные и (3) гибридные, и каждый класс проиллюстрирован тремя репрезентативными системами (1) BigIntegrator [Zhu and Risch 2011], Forward [Fu et al. 2014] и QoX [Simitsis et al. 2009, 2012]; (2) Polybase [DeWitt et al. 2013, Gankidi et al. 2014], HadoopDB [Abouzeid et al. 2009] и Estocada [Bugiotti et al. 2015];

(3) Spark SQL [Armbrust et al. 2015], BigDAWG [Gadepally et al. 2016] и CloudMdsQL [Bondiombouy et al. 2016, Kolev et al. 2016b,a]. Из других важных полихранилищ отметим Amazon Redshift Spectrum, AsterixDB, AWESOME [Dasgupta et al. 2016], Odyssey [Hacigümüs et al. 2013] и JEN [Tian et al. 2016]. В системе Odyssey используются ситуативно материализуемые представления (opportunistic materialized view), основанные на методе MISO [LeFevre et al. 2014] настройки физической структуры полихранилища.

Упражнения

Задача 11.1. Вспомните и обсудите побудительные причины для разработки NoSQL-систем и сравните их с реляционными СУБД.

Задача 11.2 (*). Объясните, в чем важность теоремы CAP. Рассмотрим распределенную архитектуру с репликацией с несколькими главными копиями (см. главу 6). Предположим, что репликация асинхронная и произошло разделение сети.

a) Какие из свойств CAP сохраняются?

b) Какой вид согласованности достигается?

Ответьте на те же вопросы в предположении, что репликация синхронная.

Задача 11.3 (**). В этой главе мы выделили четыре категории NoSQL-систем: хранилища ключей и значений, хранилища с широкими столбцами, документные хранилища и графовые базы данных.

a) Обсудите основные сходства и различия с точки зрения модели данных, языка запросов и интерфейсов к нему, архитектуры и методов реализации.

b) Для каждой категории приведите примеры, когда лучше всего использовать относящуюся к ней систему.

Задача 11.4 (**). Рассмотрим следующую упрощенную схему базы данных для ввода заказов (во вложенном реляционном формате), где подчеркнуты атрибуты, составляющие первичные ключи:

```
CUSTOMERS(CID, NAME, ADDRESS (STREET, CITY, STATE,
  COUNTRY), PHONES)
ORDERS(OID, CID, O-DATE, O-TOTAL)
ORDER-ITEMS(OID, LINE-ID, PID, QTY)
PRODUCTS(PID, P-NAME, PRICE)
```

a) Напишите соответствующие схемы для всех четырех видов NoSQL-систем. Обсудите плюсы и минусы каждого проекта с точки зрения простоты использования, администрирования базы данных, сложности запросов и производительности обновления.

b) Пусть теперь товар может быть составным, например упаковка из шести банок пива. Отразите это в схемах баз данных и обсудите последствия для каждого из четырех видов NoSQL-систем.

Задача 11.5 (**). В главе 10 было отмечено, что не существует оптимального решения задачи о разбиении графовой базы данных. Поразмышляйте о последствиях этого факта для масштабируемости графовых баз данных. Предложите обходные решения.

Задача 11.6 (**). Сравните NewSQL-систему F1 со стандартной параллельной реляционной СУБД, например MySQL Cluster, с точки зрения модели данных, языка запросов и интерфейса к нему, согласованности, масштабируемости и доступности.

Задача 11.7. Полихранилища предлагают интегрированный доступ с помощью запросов к нескольким складам данных, в т. ч. NoSQL, реляционным СУБД и HDFS. Сравните полихранилища с системами интеграции данных, описанными в главе 7.

Задача 11.8 (***). Обычно полихранилища поддерживают только запросы чтения, этого достаточно для аналитических задач. Однако по мере того как создаются все новые и новые облачные информационно емкие приложения, необходимость обновления данных в нескольких складах становится все более насущной. Поэтому появляется нужда в распределенных транзакциях. Однако транзакционные модели складов данных могут быть очень различны. В частности, большинство NoSQL-систем не поддерживают ACID-транзакции. Обсудите проблему и предложите пути ее решения.

Глава 12

Управление веб-данными

Всемирная паутина (или для краткости «веб») стала огромным репозиторием данных и документов. Хотя результаты измерений разнятся, нет сомнений, что веб растет с феноменальной скоростью[1]. Но важен не только размер, веб очень динамичен и быстро изменяется. С практической точки зрения, веб представляет собой огромный, динамичный и распределенный склад данных, и, очевидно, при доступе к веб-данным возникают проблемы управления распределенными данными.

В своей нынешней форме веб можно рассматривать как две разные, но взаимосвязанные компоненты. Первая – то, что называется *публично индексируемым вебом* (ПИВ), – включает все статические (и связанные перекрестными ссылками) веб-страницы, расположенные на веб-серверах. Их можно легко найти и проиндексировать. Вторая называется *глубинным* (или *скрытым*) вебом и состоит из огромного количества баз данных, в которых хранятся данные, скрытые от внешнего мира. Эти данные обычно извлекаются с помощью интерфейсов поиска: пользователь вводит запрос, он передается серверу базы данных, а результаты возвращаются пользователю в виде динамически сгенерированной веб-страницы. Часть глубинного веба, известная под названием «темный веб» (или «темный интернет»), состоит из зашифрованных данных, для доступа к ней необходим специальный браузер, например Tor.

Разница между ПИВ и скрытым вебом в основном заключается в способе поиска и предъявления запросов. Для поиска в ПИВ необходимо обойти страницы роботом, который переходит по ссылкам, проиндексировать собранные страницы, а затем произвести поиск в проиндексированных данных (подробнее мы обсудим это в разделе 12.2). При этом можно использовать всем известный поиск по ключевым словам или вопрос-ответную систему (QA-систему) (раздел 12.4). К скрытому вебу такой подход неприменим, потому что обойти и проиндексировать эти данные невозможно (методы поиска в скрытом вебе обсуждаются в разделе 12.5).

Есть два направления исследований по управлению веб-данными, соответствующие сообщества различны, но пересекаются. Большая часть ранних

---

[1] См. http://www.worldwidewebsize.com/.

работ по поиску в интернете и информационному поиску была посвящена поиску по ключевым словам и поисковым системам. В дальнейшем это сообщество сосредоточилось на QA-системах. Сообщество баз данных направило свои усилия на декларативные методы опроса веб-данных. Наметилась тенденция к объединению поисково-просмотрового режима доступа с декларативными запросами, но полностью потенциал здесь еще далеко не раскрыт. В 2000-х годах на сцену вышел XML как важный формат для представления и интеграции данных в вебе. Поэтому управление XML-данными вызвало значительный интерес. Хотя XML по-прежнему важен во многих прикладных областях, его использование для управления веб-данными сошло на нет из-за слишком высокой сложности. В последнее время для представления и интеграции веб-данных все чаще используют формат RDF.

Из-за того что направления исследований разошлись, не существует унифицированной архитектуры или каркаса, в рамках которого можно было бы обсуждать управление веб-данными, и приходится рассматривать разные направления по отдельности. Кроме того, полное освещение всех относящихся к вебу тем потребовало бы куда более глубокого и обширного рассмотрения, чем возможно в одной главе. Поэтому мы сосредоточимся на вопросах, напрямую связанных с управлением данными.

Начнем с обсуждения того, как можно смоделировать веб-данные в виде графа. Важны как структура этого графа, так и управление им. Это предмет раздела 12.1. Поиск в вебе обсуждается в разделе 12.2, а предъявление запросов к вебу в разделе 12.3. В разделе 12.4 кратко описываются вопросно-ответные системы, а в разделе 12.5 – поиск и опрос глубинного веба. В разделе 12.6 мы обсудим интеграцию веб-данных, сделав акцент на фундаментальные проблемы и некоторые подходы к представлению (например, веб-таблицы, XML и RDF), которые могут помочь в решении задачи.

12.1. Управление веб-графом

Веб состоит из «страниц», соединенных гиперссылками, эту структуру можно смоделировать в виде ориентированного графа, который обычно называют *веб-графом*. Его вершинами являются статические HTML-страницы, а ребрами – ссылки между страницами. Характеристики веб-графа важны для изучения проблем управления данными, поскольку эта структура используется при поиске в вебе, классификации и категоризации веб-контента и решении других относящихся к вебу задач. Кроме того, RDF-представление, обсуждаемое в разделе 12.6.2.2, формализует веб-граф с применением точно определенной нотации. Перечислим важные характеристики веб-графа:

a) он в высшей степени изменчив. Мы уже обсуждали скорость, с которой растет веб-граф. К тому же значительная доля страниц часто обновляется;
b) он разрежен. Граф называется разреженным, если средняя степень его вершин много меньше количества вершин. Это означает, что у каждой вершины мало соседей, пусть даже все вершины как-то связаны. Из

разреженности веб-графа вытекает интересная особенность его формы, которую мы обсудим чуть ниже;

c) он «самоорганизующийся». Веб содержит множество сообществ, каждое из которых состоит из страниц, посвященных определенной теме. Эти сообщества организуются сами, без какого-либо «централизованного управления», и приводят к появлению четко очерченных подграфов в веб-графе;

d) это граф типа «мир тесен». Данное свойство связано с разреженностью – у каждой вершины графа, возможно, немного соседей (т. е. ее степень мала), но многие вершины связаны через промежуточные. Графы типа «мир тесен» впервые были выявлены в общественных науках, когда было замечено, что незнакомые между собой люди часто имеют цепочку общих знакомых. Это верно и для веб-графа;

e) это степенной граф. Распределение входящих и исходящих степеней вершин графа подчиняется степенному закону. Это означает, что вероятность того, что вершина имеет входящую (исходящую) степень i, пропорциональна $1/i^{\alpha}$ для некоторого $\alpha > 1$. Значение α приближенно равно 2.1 для входящих степеней и 7.2 для исходящих.

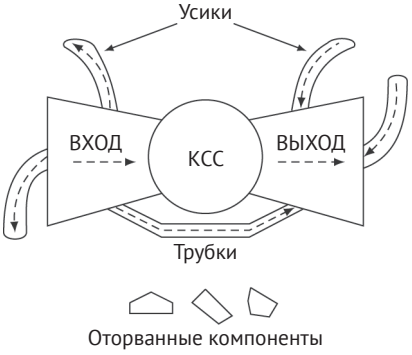


Рис. 12.1 ❖ Структура веб-графа, имеющего форму галстука-бабочки (из работы [Kumar et al. 2000])

Это подводит нас к обсуждению строения веб-графа, который имеет форму «галстука-бабочки» (рис. 12.1). В нем имеется компонента сильной связности (узел в центре), в которой между каждой парой страниц имеется путь. Приведенные ниже числа взяты из исследования 2000 года; они могли измениться, но изображенная на рисунке форма сохранилась. Эти числа следует рассматривать как относительные, а не абсолютные размерные показатели. В компоненту сильной связности (КСС) попадает приблизительно 28 % веб-страниц. Еще 21 % страниц попадают в компоненту «ВХОД»: для каждой из них имеется путь к страницам, принадлежащим КСС, но нет пути из КСС к этой странице. Аналогично, в компоненту «ВЫХОД» попадают страницы, на которые можно попасть из КСС, но нельзя вернуться; таких тоже приблизительно 21 %. «Усики» состоят из страниц, на которые нельзя попасть из КСС и из которых нет путей в КСС. Таких примерно 22 %. Это страницы, которые еще не «открыты» и не имеют связей с более хорошо известными частями веба. Наконец, существуют оторванные компоненты – на них не ссылается никто, кроме членов их малого сообщества, и сами они больше ни на кого не ссылаются. Они составляют около 8 % веба. Эта структура интересна тем, что определяет результаты, получаемые при поиске в вебе. Кроме того, она отличается от структуры типичных изучаемых графов и требует разработки специальных алгоритмов управления.

12.2. Поиск в вебе

Поиск в вебе (или веб-поиск) подразумевает нахождение «всех» страниц, релевантных (т. е. содержащих материал, соответствующий теме запроса) ключевым словам, заданным пользователем. Естественно, найти все страницы невозможно, нельзя даже сказать, все ли страницы найдены, поскольку поиск производится в базе данных страниц, которые были собраны и проиндексированы. Обычно существует несколько страниц, релевантных запросу, поэтому они предъявляются пользователю отсортированными по релевантности, и этот порядок определяется поисковой системой.

Абстрактная архитектура общей поисковой системы показана на рис. 12.2. Ниже мы обсудим ее компоненты.

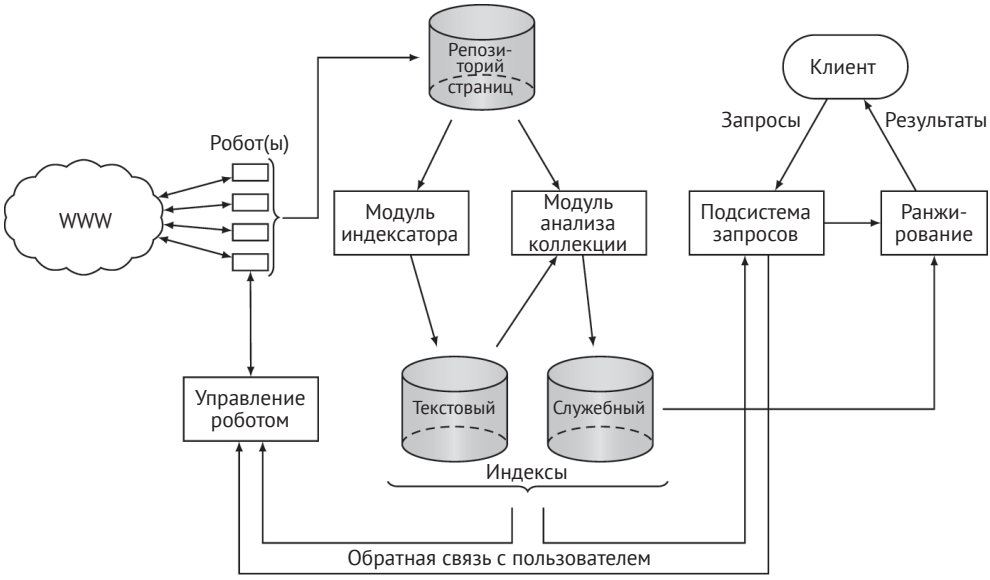


Рис. 12.2 ❖ Архитектура поисковой системы (из работы [Arasu et al. 2001])

В любой поисковой системе существует *робот* (crawler), на который возлагается одна из самых важных ролей. Он просматривает веб от имени поисковой системы и собирает сведения о веб-страницах. Роботу задается начальное множество страниц, точнее унифицированных локаторов ресурсов (URL), определяющих местонахождение этих страниц. Робот считывает и разбирает страницы, выделяет встречающиеся в них URL и помещает их в очередь. На следующей итерации цикла робот выбирает URL из очереди (в определенном порядке) и читает соответствующую страницу. Этот процесс повторяется, пока робот не остановится. Управляющий модуль решает, какой URL посетить следующим. Прочитанные страницы сохраняются в репозитории. В разделе 12.2.1 работа робота описана более подробно.

Модуль индексатора отвечает за построение индексов по страницам, загруженным роботом. Можно построить много разных индексов, но самые известные – *текстовые индексы* и *индексы ссылок*. Чтобы создать текстовый индекс, индексатор строит огромную «справочную таблицу», из которой можно получать URL всех страниц, на которых встречается данное слово. Индекс ссылок описывает ссылочную структуру веба и дает информацию о входящих и исходящих ссылках для каждой страницы. В разделе 12.2.2 объясняется современная технология индексирования с упором на способы эффективного хранения индексов.

Модуль ранжирования отвечает за сортировку большого числа результатов, так чтобы те, что считаются наиболее релевантными запросу пользователя, оказались в начале списка. Задача ранжирования вызвала повышенный интерес в связи с необходимостью выйти за рамки традиционных методов информационного поиска и учесть особые характеристики веба – веб-запросы обычно малы, но выполняются над очень большим объемом данных. В разделе 12.2.3 приводится введение в алгоритмы ранжирования и описываются подходы, позволяющие задействовать ссылочную структуру веба для улучшения результатов ранжирования.

12.2.1. Обход веба роботом

Как было сказано выше, робот просматривает веб от имени поисковой системы, чтобы извлечь информацию о посещенных страницах. Учитывая размер веба, изменчивость страниц и ограниченные вычислительные ресурсы и память роботов, понятно, что обойти весь веб невозможно. Поэтому робот должен быть устроен так, чтобы посещать «самые важные» страницы раньше прочих. Следовательно, проблема состоит в том, чтобы посещать страницы в порядке важности.

При проектировании робота нужно ответить на много вопросов. Поскольку основная цель робота – посещать важные страницы раньше остальных, встает вопрос, как определить важность страницы. Для этого нужен показатель, измеряющий важность данной страницы. Показатель может быть статическим, т. е. не зависеть от количества запросов на поиск этой страницы, или динамическим – учитывающим запросы. Примером статического показателя является количество ссылок, указывающих на страницу P_i (*обратных ссылок*). Можно дополнительно принять во внимание важность ссылающихся страниц, как в популярной метрике PageRank, используемой Google и другими системами. Пример динамического показателя – сходство текста страницы P_i с запросом, вычисляемое с помощью хорошо известных метрик информационного поиска.

Мы познакомились с PageRank в главе 10 (пример 10.4). Напомним, что ранг страницы P_i, обозначаемый $PR(P_i)$, – это просто сумма нормированных рангов всех ссылающихся на P_i страниц P_j (их множество обозначается B_{P_i}), где нормировочный коэффициент для страницы P_j равен количеству имеющихся на ней ссылок:

$$PR(P_i) = \sum_{P_j \in B_{P_i}} \frac{PR(P_j)}{|F_{P_j}|}.$$

Эта формула подсчитывает ранг страницы на основе количества обратных ссылок на нее, но нормирует вклад каждой такой страницы P_j на число ссылок с нее. Идея в том, что важнее ссылки со страниц, которые содержат не слишком много хорошо продуманных ссылок, а не указывают на все что ни попадя, но тогда вклад страницы нужно нормировать на количество страниц, на которые она ссылается.

Второй вопрос: как робот решает, какую следующую страницу посетить, после того как посетил конкретную страницу? Мы уже упоминали, что робот ведет очередь, в которой хранит URL обнаруженных страниц, и добавляет в нее элементы в процессе анализа каждой страницы. Поэтому проблема в том, как упорядочить эту очередь. Тут возможны разные стратегии. Например, можно посещать URL в том порядке, в каком они встречались; это называется *обход в ширину*. Другой вариант – случайное упорядочение, когда робот выбирает случайный URL из очереди ожидающих посещения страниц. Можно также использовать метрики, сочетающие упорядочение с описанным выше ранжированием по важности, например счетчиком обратных ссылок или PageRank.

Рассмотрим, как можно использовать PageRank для этой цели. Нам понадобится немного изменить приведенную выше формулу. Теперь мы моделируем случайного посетителя: дойдя до страницы P, случайный посетитель либо выбирает один из встречающихся на ней URL с вероятностью d, либо переходит на какую-то случайную страницу с вероятностью $1 - d$. Тогда модифицированная формула PageRank выглядит так:

$$PR(P_i) = (1 - d) + d \sum_{P_j \in B_{P_i}} \frac{PR(P_j)}{|F_{P_j}|}.$$

Упорядочение URL по этой формуле позволяет учесть важность страницы при определении порядка посещения. Иногда первый член нормируют на общее число страниц в вебе.

Пример 12.1. Рассмотрим веб-граф на рис. 12.3, в котором каждая веб-страница P_i представлена вершиной и из P_i в P_j ведет ориентированное ребро, если на P_i встречается ссылка на P_j. В предположении, что берется общепринятое значение $d = 0.85$, PageRank страницы P_2 равен $PR(P_2) = 0.15 + 0.85(PR(P_1)/2 + PR(P_3)/3)$. Чтобы произвести вычисления по этой рекурсивной формуле, мы сначала присваиваем всем страницам одинаковые значения PageRank (в данном случае 1/6, потому что всего имеется 6 страниц), а затем вычисляем $PR(P_i)$, пока алгоритм не сойдется (т. е. значения перестанут изменяться). ♦

Поскольку многие веб-страницы со временем изменяются, обход веба происходит непрерывно, и страницы посещаются снова и снова. Вместо того чтобы каждый раз начинать обход с нуля, лучше избирательно посещать некоторые страницы и обновлять собранную ранее информацию. Роботы,

которые так и поступают, называются *инкрементными*. Они стараются, чтобы информация в их репозиториях была по возможности актуальной. Инкрементный робот решает, какие страницы посетить повторно, исходя из частоты изменения страницы, или просто производит выборку. В подходах *на основе частоты изменения* для определения того, насколько часто нужно заходить на страницу, используется оценка частоты изменения страницы. Интуитивно кажется, что часто изменяющиеся страницы следует посещать чаще, но это не всегда верно – любая информация, извлеченная из такой страницы, с большой вероятностью скоро устареет, поэтому иногда лучше увеличить интервал между ее посещениями. Можно также разработать адаптивный инкрементный робот такой, что на порядок обхода в очередном цикле влияет информация, собранная в предыдущем цикле. В подходах *на основе выборки* акцент делается на сайтах, а не на отдельных страницах. Чтобы оценить, насколько изменился сайт, производится небольшая выборка из множества его страниц, и на основе ее анализа принимается решение, как часто посещать этот сайт.

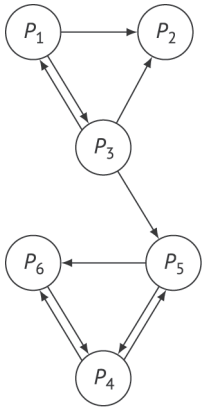


Рис. 12.3 ❖ Представление веб-графа для вычисления PageRank

Некоторые поисковые системы специализируются на поиске страниц, относящихся к определенной теме. В них используются *сфокусированные роботы*, оптимизированные для этой темы. Такой робот ранжирует страницы на основе релевантности данной теме и использует полученные оценки для решения о том, какую страницу посетить следующей. Для оценки релевантности применяются методы, широко распространенные в информационном поиске, а для определения темы страницы – методы машинного обучения. Эти методы выходят за рамки книги, но отметим, что их немало, например наивный байесовский классификатор и его обобщения, обучение с подкреплением и другие.

Для достижения масштабируемости обход можно распараллелить, запустив *параллельные роботы*. В любом проекте параллельных роботов необходимо предусмотреть минимизацию накладных расходов на распараллелива-

ние. Например, два работающих параллельно робота могут загрузить один и тот же набор страниц. Очевидно, что такое пересечение желательно предотвратить, для чего действия роботов нужно координировать. *Центральный координатор* динамически назначает каждому роботу набор страниц. Другая схема координации подразумевает логическое разбиение веба на разделы. Каждый робот знает о своем разделе, поэтому в центральном координаторе нет необходимости. Эта схема называется *статическим назначением*.

12.2.2. Индексирование

Для эффективного поиска над страницами, проанализированными роботом, строится ряд индексов, показанных на рис. 12.2. Наиболее важны два: *структурный индекс* (или *индекс ссылок*) и *текстовый индекс* (или *индекс по содержимому*).

12.2.2.1. Структурный индекс

Структурный индекс основан на графовой модели, рассмотренной в разделе 12.1, в которой граф представляет структуру посещенной роботом части веба. Очень важно организовать эффективное хранение и поиск этих страниц, для чего разработано два метода, описанных в разделе 12.1. Структурный индекс можно использовать для получения важной информации о ссылках между страницами, например об *окрестности* страницы и о страницах, расположенных на одном уровне с ней.

12.2.2.2. Текстовый индекс

Самым важным и наиболее часто используемым является *текстовый индекс*. Индексы для поддержки текстового поиска можно реализовать с помощью любого из методов доступа, которые традиционно применяются для поиска в наборах документов, например: *суффиксные массивы*, *инвертированные файлы* или *индексы* и *файлы сигнатур*. Полное рассмотрение всех этих индексов выходит за рамки книги, но использование инвертированных индексов мы все же обсудим, поскольку они наиболее популярны в этом контексте.

Инвертированный индекс представляет собой набор инвертированных списков, каждый список ассоциирован с одним словом. В общем случае элементами инвертированного списка для данного слова являются идентификаторы документов, в которых это слово встречается. В инвертированном списке можно также сохранить местоположение слова на странице. Эта информация нужна для поиска с учетом близости и для ранжирования результатов запроса. Алгоритмы поиска часто используют также дополнительную информацию о термах, встречающихся на странице. Например, термам, набранным полужирным шрифтом (окруженным тегами `<strong>`), встретившимся в заголовках (внутри HTML-тегов `<H1>` или `<H2>`) или в текстах ссылок, алгоритмы ранжирования могут назначать больший вес.

Помимо инвертированного списка, во многих текстовых индексах хранится *лексикон* – список всех термов, встречающихся в индексе. Дополнительно

лексикон может содержать статистику, связанную с термами, которая используется алгоритмами ранжирования.

Построение и сопровождение индекса сталкиваются с тремя основными трудностями:

1) в общем случае для построения инвертированного индекса нужно обработать каждую страницу – прочитать все встречающиеся на ней слова и запомнить местоположение каждого слова. В конце этой процедуры инвертированные файлы записываются на диск. Для небольших статических наборов эта задача тривиальна, но становится весьма сложной, если нужно обрабатывать такой большой и динамически изменяющийся набор, как веб;
2) в связи с быстрым изменением веба возникает вопрос, как обеспечить актуальность индекса. В предыдущем разделе мы говорили, что для этой цели следует использовать инкрементные роботы, но все равно индексы необходимо периодически перестраивать, потому что большинство известных методов инкрементного обновления плохо работают, когда объем изменений велик, как, например, между двумя соседними обходами веба;
3) форматы хранения инвертированного индекса необходимо тщательно продумывать. Необходимо найти компромисс между выигрышем от использования сжатого индекса, что позволяет кешировать части индекса в памяти, и накладными расходами на распаковку во время выполнения запроса. Достижение приемлемого баланса становится серьезной проблемой при работе с наборами документов масштаба веба.

Чтобы решить эти проблемы и получить хорошо масштабируемый текстовый индекс, мы можем распределить его, построив либо *локальный инвертированный индекс* на каждой машине, где работает поисковая система, либо *глобальный инвертированный индекс*, который разделяется всеми машинами. Мы не станем дальше развивать эту тему, поскольку похожие вопросы управления распределенными данными и каталогом уже рассматривались в предыдущих главах.

12.2.3. Ранжирование и анализ ссылок

Типичная поисковая система возвращает много страниц, предположительно релевантных запросу. Однако эти страницы различаются по качеству и степени релевантности. Никто не хочет, чтобы пользователь просматривал весь этот большой набор в поисках высококачественной страницы. Нужен алгоритм ранжирования, который помещал бы более интересные пользователю страницы в начало списка.

Для ранжирования набора страниц можно воспользоваться *алгоритмами на основе ссылок*. Повторяя сказанное выше, напомним, что если страница P_j содержит ссылку на страницу P_i, то, надо полагать, авторы P_j считали страницу P_i достойной. Поэтому страница, на которую ведет много ссылок, вероятно, хорошего качества, а значит, количество входящих ссылок можно

рассматривать как критерий ранжирования. Это интуитивное соображение лежит в основе алгоритмов ранжирования, но, конечно, каждый алгоритм формализует его по-своему. Мы уже обсуждали алгоритм PageRank, который используется не только для обхода веба, но и для ранжирования. А сейчас обсудим еще один алгоритм, HITS, чтобы проиллюстрировать разные подходы к решению проблемы.

Алгоритм HITS также основан на ссылках. В нем используются понятия «авторитетных страниц» и «хаб-страниц». Хорошая авторитетная страница получает высокий ранг. Между авторитетными страницами и хабами имеется взаимно усиливающая связь: хорошей авторитетной страницей считается та, на которую ссылаются много хороших хабов, а хорошей хаб-страницей – та, которая ссылается на много авторитетных страниц. Поэтому страница, на которую указывает много хабов (хорошая авторитетная страница), вероятно, имеет высокое качество.

Начнем с веб-графа $G = (V, E)$, где V – множество страниц, а E – множество связей между ними. Каждой странице P_i в V назначается два неотрицательных веса (a_{P_i}, h_{P_i}), представляющих ее авторитетность и ценность как хаба. Эти значения обновляются следующим образом. Если на страницу P_i указывает много хороших хаб-страниц, то a_{P_i} увеличивается, чтобы отразить все ссылающиеся на нее страницы P_j (нотация $P_j \to P_i$ означает, что на странице P_j есть ссылка на P_i):

$$a_{P_i} = \sum_{\{P_j | P_j \to P_i\}} h_{P_j};$$

$$h_{P_i} = \sum_{\{P_j | P_j \to P_i\}} a_{P_j}.$$

Таким образом, авторитетность (ценность хаба) страницы P_i равна сумме ценностей хаба (авторитетностей) страниц, ссылающихся на P_i.

12.2.4. Поиск по ключевым словам

Системы поиска по ключевым словам – самые популярные инструменты поиска информации в вебе. Они просты и позволяют задавать нечеткие вопросы, на которые может не быть точного ответа, но имеются приближенные ответы, содержащие факты, «похожие» на ключевые слова. Однако существуют очевидные ограничения на возможности простого поиска по ключевым словам. Прежде всего такого поиска недостаточно, чтобы выразить сложные запросы. Частично эту проблему можно решить с помощью итеративных запросов, когда ответ на предыдущий запрос пользователя является контекстом для следующих запросов от него. Второе ограничение заключается в том, что поиск по ключевым словам не использует глобальное представление об информации в вебе в том смысле, в каком запросы к базе данных используют информацию, содержащуюся в схеме. Конечно, можно возразить, что для веб-данных само понятие схемы не имеет смысла, но тем не менее отсутствие глобального представления о данных является проблемой. Третья проблема связана с возможными ошибками в интерпретации намерений

пользователя – неправильный выбор ключевых слов может привести к выдаче большого числа нерелевантных результатов.

Поиск по категории решает одну из проблем поиска по ключевым словам, а именно отсутствие глобального представления о вебе. Поиск по категории имеет много названий: веб-каталог, желтые страницы или тематический справочник. Существует несколько публичных веб-каталогов, например World Wide Web Virtual Library (http://vlib.org)[1]. Веб-каталог представляет собой иерархическую классификацию человеческих знаний. Хотя классификатор представлен в виде дерева, на самом деле это ациклический ориентированный граф, связанный перекрестными ссылками.

Если целью выступает категория, то веб-каталог является полезным инструментом. Однако не все страницы классифицированы, поэтому использовать каталог для поиска не всегда возможно. Кроме того, для классификации веб-страниц естественный язык не является стопроцентно эффективным средством. Для оценки подаваемых на рассмотрение страниц нужны люди, так что этот подход не слишком рентабелен и плохо масштабируется. Наконец, некоторые страницы со временем изменяются, поэтому поддержание актуальности каталога сопряжено со значительными издержками.

Предпринимались также попытки использовать несколько поисковых систем для улучшения точности и полноты ответа на запрос. Метапоисковик – это веб-служба, которая принимает запрос от пользователя и посылает его нескольким различным поисковым системам. Затем метапоисковик собирает ответы и возвращает пользователю объединенный результат. Он умеет сортировать результаты по разным атрибутам, например по владельцу, ключевому слову, дате или популярности. Примерами могут служить системы Dogpile (http://www.dogpile.com/), MetaCrawler (http://www.metacrawler.com/) и IxQuick (http://www.ixquick.com/). Разные метапоисковики применяют различные способы объединения результатов и трансляции запросов пользователей на язык конкретной поисковой системы. Пользователь может обратиться к метапоисковику с помощью клиентской программы или на сайте. Каждая поисковая система покрывает небольшую часть веба. Цель метапоисковика – охватить больше страниц, чем одна поисковая система, объединив результаты поиска.

12.3. Запросы к вебу

Декларативные запросы и методы их эффективного выполнения всегда находились в центре внимания разработчиков технологий баз данных. Было бы хорошо, если бы удалось применить методы баз данных к вебу. Тогда доступ к вебу можно было бы рассматривать как отдаленный аналог доступа к большой базе данных. В этом разделе мы обсудим несколько предложений такого рода.

При переносе традиционных идей опроса баз данных на веб-данные мы сталкиваемся с несколькими трудностями. Самая важная, пожалуй, состоит в том, что опрос базы данных предполагает наличие строгой схемы. Выше

[1] Список таких каталогов имеется на странице https://en.wikipedia.org/wiki/List_of_web_directories.

мы отмечали, что для веб-данных говорить о такой схеме не приходится[1]. В лучшем случае веб-данные *слабо структурированы* – у данных может быть какая-то структура, но, в отличие от баз данных, она нежесткая, нерегулярная и неполная, поэтому разные экземпляры данных могут быть структурно похожи, но не идентичны (какие-то атрибуты могут отсутствовать, другие, наоборот, добавляются, структура может изменяться). Очевидно, что опросу бессхемных данных некоторые трудности внутренне присущи.

Вторая проблема заключается в том, что веб – не просто собрание слабо структурированных данных (и документов). Между сущностями, представленными веб-данными (например, страницами), имеются ссылки, которые необходимо учитывать. Как и при поиске, который мы обсуждали в предыдущем разделе, при выполнении запросов к вебу следование по ссылкам может принести ощутимую пользу. А значит, ссылки следует рассматривать как полноправные объекты.

Третья трудность – отсутствие общепринятого языка запроса к веб-данным, аналогичного SQL. Выше мы отмечали, что для поиска по ключевым словам используется очень простой язык, но его недостаточно для выражения более сложных запросов к веб-данным. Постепенно вырабатывается общее мнение относительно базовых конструкций такого языка (например, путевые выражения), но никакого стандарта нет. Однако появились стандартизованные языки для таких моделей данных, как XML и RDF (XQuery для XML и SPARQL для RDF). Мы отложим их обсуждение до раздела 12.6, посвященного интеграции веб-данных.

12.3.1. Веб как слабо структурированные данные

Один из подходов к опросу веб-данных состоит в том, чтобы рассматривать веб как собрание слабо структурированных данных. Были разработаны модели и языки для опроса таких данных. Изначально они предназначались не для работы с веб-данными, а для удовлетворения требований к растущим наборам данных, не имеющим такой строгой схемы, как в реляционном случае. Но поскольку подобные характеристики присущи и веб-данным, оказалось, что эти модели и языки применимы и в этой области. Мы продемонстрируем данный подход на примере модели OEM и языка Lorel; другие системы, например UnQL, похожи.

OEM (Object Exchange Model – модель обмена объектами) – это самоописываемая слабо структурированная модель данных. Под самоописанием понимается, что каждый объект определяет схему, которой он следует.

Объект OEM определяется как четверка ⟨`label`, `type`, `value`, `oid`⟩, где `label` – строка символов, описывающая, что именно представляет объект, `type` – тип значения объекта, `value` – значение, а `oid` – идентификатор, отличающий данный объект от всех остальных. Тип объекта может принимать значения `atomic` – тогда объект называется *атомарным*, или `complex` – тогда объект называется *составным*. Атомарный объект имеет значение примитивного типа,

[1] Мы сейчас говорим от «открытом» вебе; у данных, скрытых в глубинном вебе, схема, возможно, есть, но пользователям она обычно недоступна.

например целое, вещественное или строку, а составной включает другие объекты, которые сами могут быть атомарными или составными. Значением составного объекта является набор идентификаторов объектов.

Пример 12.2. Рассмотрим библиографическую базу данных, состоящую из набора документов. Выдержка из такой базы показана на рис. 12.4. В каждой строке присутствует один объект OEM, а отступы проявляют структуру объекта. Например, во второй строке `<doc, complex, &o3, &o6, &o7, &o20, &o21, &o2>` определен объект с меткой `doc`, типом `complex`, идентификатором `&o2`, состоящий из объектов с идентификаторами `&o3`, `&o6`, `&o7`, `&o20` и `&o21`.

```
<bib, complex, {&o2, &o22, &034}, &o1>
   <doc, complex, {&o3, &o6, &o7, &o20, &o22}, &o2>
      <authors, complex, {&o4, &o5}, &o3>
         <author, string, "M. Tamer Ozsu", &o4>
         <author, string, "Patrick Valduriez", &o5>
      <title, string, "Principles of Distributed ...", &o6>
      <chapters, complex, {&o8, &o11, &o14, &o9}, &o7>
         <chapter, complex, {&o9, &o10}, &o8>
            <heading, string, "...", &o9>
            <body, string, "...", &o10>
            ...
         <chapter, complex, {&o18, &o19}, &9>
            <heading, string, "...", &o18>
            <body, string, "...", &o19>
      <what, string, "Book", &o20>
      <price, float, 98.50, &o21>
   <doc, complex, {&o23, &o25, &o26, &o27, &o28}, &o22>
      <authors, complex, {&o24, &o4}, &o23>
         <author, string, "Yingying Tao", &o24>
      <title, string, "Mining data streams ...", &o25>
      <venue, string, "CIKM", &o26>
      <year, integer, 2009, &o27>
      <sections, complex, {&o29, &o30, &o31, &o32, &o33}, &28>
         <section, string, "...", &o29>
           ...
         <section, string, "...", &o33>
   <doc, complex, {&o16,&o9,&o7,&o18,&o19,&o20,&o21},&o34>
      <author, string, "Anthony Bonato", &o35>
      <title, string, "A Course on the Web Graph", &o36>
      <what, string, "Book", &o20>
      <ISBN, string, "TK5105.888.B667", &o37>
      <chapters, complex, {&o39, &o42, &o45}, &o38>
         <chapter, complex, {&o40, &o41}, &o39>
            <heading, string, "...", &o40>
            <body, string, "...", &o41>
         <chapter, complex, {&o43, &o44}, &o42>
            <heading, string, "...", &o43>
            <body, string, "...", &o44>
         <chapter, complex, {&o46, &o47}, &45>
            <heading, string, "...", &o46>
            <body, string, "...", &o47>
      <publisher, string, "AMS", &o48>
```

Рис. 12.4 ❖ Пример спецификации в OEM

Эта база данных содержит три документа: (&o2, &o22, &o34); первый и третий – книги, второй – статья. У двух книг (и даже у книг и статьи) есть общие черты, но есть и различия. Например, для документа &o2 задана цена, а для &o34 – ISBN и сведения об издательстве, которые для &o2 отсутствуют. ♦

Как уже было сказано, данные в OEM самоописываемые, т. е. каждый объект описывает себя с помощью типа и метки. Легко видеть, что данные можно представить в виде графа с помеченными вершинами, в котором вершины соответствуют объектам OEM, а ребра ведут от объектов к подобъектам. Однако в литературе принято моделировать данные графом с помеченными ребрами: если объект o_j является подобъектом o_i, то метка o_j присваивается ребру, соединяющему o_i с o_j, а метками вершин служат идентификаторы объектов. В примере 12.3 мы используем именно такое представление с помеченными вершинами и ребрами.

Пример 12.3. На рис. 12.5 изображено представление графа с помеченными вершинами и ребрами для базы данных OEM из примера 12.2. Обычно каждая терминальная вершина (из которой не исходит ни одного ребра) содержит также значение объекта. Но чтобы упростить картинку, мы опустили значение. ♦

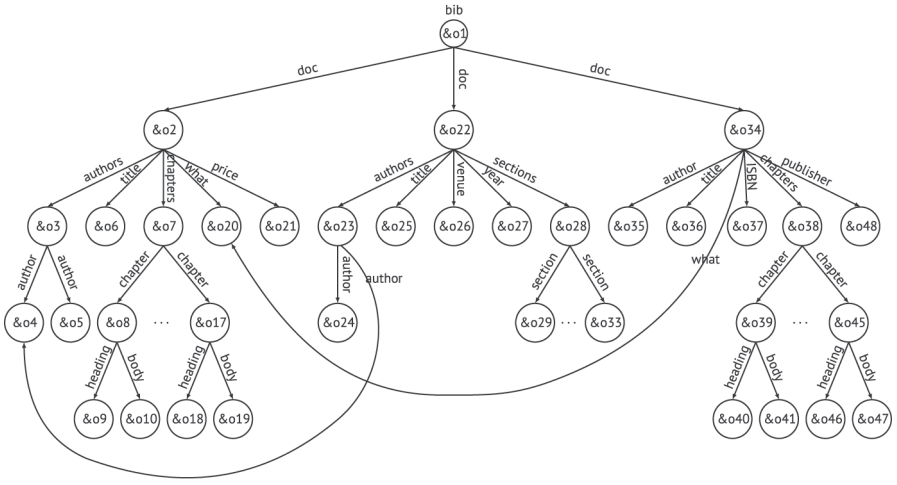


Рис. 12.5 ❖ Граф, соответствующий базе данных OEM из примера 12.2

Идея слабой структурированности хорошо ложится на моделирование веб-данных, поскольку допускает представление в виде графа. К тому же она применима к данным, которые имеют какую-то структуру, но не такую жесткую, полную и регулярную, как в традиционных базах данных. Пользователям, запрашивающим данные, не нужно знать о полной структуре, поэтому и для выражения запроса не следует требовать наличия такой информации. Представления данных из каждого источника в виде графов генерируются обертками, которые мы обсуждали в разделе 7.2.

Для опроса слабо структурированных данных разработано несколько языков. Мы ограничимся рассмотрением только одного из них, Lorel, другие похожи на него.

В Lorel принята знакомая структура SELECT-FROM-WHERE, но во фразах SELECT, FROM и WHERE допускаются путевые выражения. *Путевое выражение* вообще является основной конструкцией в запросах на Lorel. В простейшей форме это последовательность меток, начинающаяся именем объекта или переменной, обозначающей объект. Например, bib.doc.title – путевое выражение, которое интерпретируется следующим образом: начать с bib, идти по ребру с меткой doc, а затем по ребру с меткой title. На рис. 12.5 есть три пути, отвечающих этому выражению: (i) &o1.doc:&o2.title:&o6, (ii) &o1.doc:&o22.title:&o25, (iii) &o1.doc:&o34.title:&o36. Каждый из них называется *путем к данным*. В Lorel путевые выражения могут быть более сложными регулярными выражениями, за именем объекта или переменной может следовать не только метка, но и более общее выражение, включающее конъюнкцию, дизъюнкцию (|), итерацию (? означает 0 или 1 повторение, + – 1 или более, * – 0 или более) и метасимволы (#).

Пример 12.4. Ниже приведены примеры допустимых путевых выражений на языке Lorel:

a) bib.doc(.authors)?.author : начать с bib, идти по ребру doc и по ребру author, между которыми может находиться необязательное ребро authors;
b) bib.doc.#.author : начать с bib, идти по ребру doc, затем может встретиться сколь угодно много ребер с любыми метками (описываются метасимволом #), но закончить следует проходом по ребру author;
c) bib.doc.%price : начать с bib, идти по ребру doc, затем по ребру, метка которого содержит строку «price», которой могут предшествовать какие-то символы. ♦

Пример 12.5. Ниже приведены запросы на языке Lorel, в которых используются некоторые путевые выражения из примера 12.4.

a) Найти названия документов, автором которых является Patrick Valduriez.

```
SELECT D.title
FROM   bib.doc D
WHERE  bib.doc(.authors)?.author = "Patrick Valduriez"
```

В этом запросе фраза **FROM** ограничивает область просмотра документами (doc), а фраза **SELECT** задает узлы, достижимые из документов путем прохода по ребру с меткой title. Можно было бы записать предикат **WHERE** в виде

```
D(.authors)?.author = "Patrick Valduriez".
```

b) Найти авторов всех книг, цена которых меньше 100 долларов.

```
SELECT D(.authors)?.author
FROM bib.doc D
WHERE D.what = "Books" AND D.price < 100
```

♦

Слабо структурированный подход к моделированию и опросу веб-данных простой и гибкий. Он предлагает естественный способ описания иерархической структуры веб-объектов, а значит, до некоторой степени поддерживает ссылочную структуру веб-страниц. Но у него есть и недостатки. Модель данных слишком проста – она не подразумевает какую-либо внутреннюю структуру записи (каждая вершина является простой сущностью) и не поддерживает упорядоченность, поскольку на множестве вершин OEM-графа нет никакого отношения порядка. Кроме того, поддержка ссылок рудиментарна, т. к. ни модель, ни язык не различают ссылки разных типов, хотя они могут выражать как отношение «является частью» между объектами, так и связи между различными сущностями, соответствующими вершинам. Однако эти различия невозможно смоделировать, как невозможно сформулировать запрос с указанием типа ссылки.

Наконец, структура графа может стать весьма сложной, что затрудняет формулировку запросов. Хотя Lorel предлагает ряд средств (в частности, метасимволы), чтобы упростить запросы, приведенные выше примеры показывают, что пользователю все равно нужно знать общую структуру слабо структурированных данных. OEM-графы больших баз данных сложны, и пользователям трудно выписывать путевые выражения. Таким образом, требуется как-то «сжать» граф, чтобы получилось описание типа схемы разумного размера, которое могло бы помочь при написании запросов. Для этого предложена конструкция, называемая DataGuide (гид по данным). DataGuide представляет собой граф, в котором каждый путь, присутствующий в соответствующем OEM-графе, встречается только один раз. Он динамичен в том смысле, что когда OEM-граф изменяется, соответствующий ему DataGuide тоже обновляется. Таким образом, мы получаем лаконичный и точный конспект слабо структурированной базы данных, которую можно использовать как облегченную схему, полезную для изучения структуры базы данных, формулирования запросов, хранения статистической информации и оптимизации запросов.

Пример 12.6. DataGuide, соответствующий OEM-графу из примера 12.3, показан на рис. 12.6. ♦

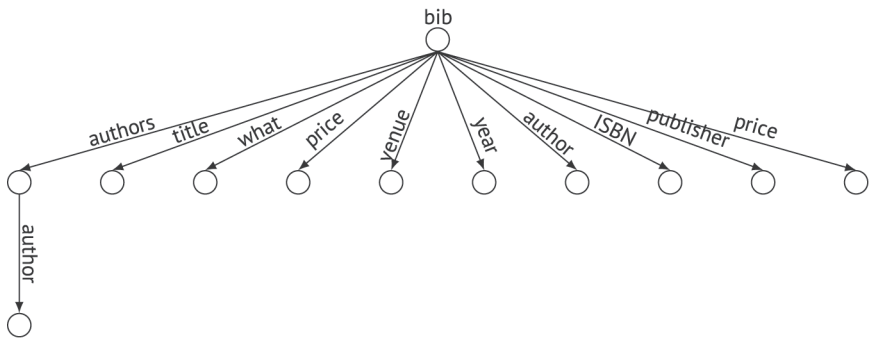


Рис. 12.6 ❖ Гид DataGuide, соответствующий OEM-графу из примера 12.3

12.3.2. Языки веб-запросов

Цель подходов из этой категории – напрямую опрашивать характеристики веб-данных, уделяя особое внимание надлежащей обработке *ссылок*. Отправной точкой является стремление преодолеть недостатки поиска по ключевым словам, предложив абстракции, правильно улавливающие как структуру документов (как в слабо структурированных подходах), так и внешние ссылки. Они объединяют запросы к содержимому (например, по ключевым словам) и структурные запросы (например, путевые выражения).

Специально для работы с веб-данными создано несколько языков, которые можно отнести к первому и второму поколениям. Языки первого поколения моделируют веб как набор взаимосвязанных *атомарных* объектов. Следовательно, на них можно выразить запросы, которые ищут объекты в вебе по структуре ссылок и текстовому содержимому, но не запросы, в которых учитывается внутренняя структура этих объектов. Языки второго поколения моделируют веб как набор взаимосвязанных *структурных* объектов, а значит, позволяют выражать запросы к внутренней структуре документа как в слабо структурированных языках. К языкам первого поколения относятся WebSQL, W3QL и WebLog, а к языкам второго поколения – WebOQL и StruQL. Мы продемонстрируем общие идеи на примере языков WebSQL и WebOQL.

WebSQL – один из ранних языков, в котором сочетается поиск и навигация. Он напрямую запрашивает хранящиеся в веб-документах данные (обычно в формате HTML), которые содержат определенный контент и могут включать ссылки на другие страницы или другие объекты (например, PDF-файлы или изображения). Ссылки рассматриваются как полноправные объекты, причем выделяется несколько типов ссылок, которые мы обсудим ниже. Как и раньше, структуру можно представить графом, но WebSQL запоминает информацию о веб-объектах в двух *виртуальных* отношениях:

```
DOCUMENT(URL, TITLE, TEXT, TYPE, LENGTH, MODIF)
LINK(BASE, HREF, LABEL)
```

В отношении `DOCUMENT` хранится информация о каждом веб-документе. Поле `URL` идентифицирует веб-объект и является первичным ключом отношения, `TITLE` содержит название веб-страницы, `TEXT` – ее текстовое содержимое, `TYPE` – тип веб-объекта (HTML-документ, изображение и т. д.), `LENGTH` – длину, а `MODIF` – дату последней модификации объекта. Все атрибуты, кроме `URL`, могут принимать значение `null`. В отношении `LINK` запоминается информация о ссылках. Поле `BASE` содержит URL того HTML-документа, в котором ссылка встречается, `HREF` – URL документа, на который ссылка ведет, а `LABEL` – метку ссылки.

Язык запросов WebSQL напоминает SQL, дополненный путевыми выражениями, более выразительными, чем в Lorel; в частности, они различают типы ссылок:

a) *внутренняя ссылка* в пределах одного документа (#>);
b) *локальная ссылка* между документами на одном сервере (->);
c) *глобальная ссылка* на документ, расположенный на другом сервере (=>);
d) *пустой путь* (=).

Эти типы ссылок образуют алфавит путевых выражений. Из них и обычных конструкторов регулярных выражений можно формировать различные пути, как показано в примере 12.7.

Пример 12.7. Ниже приведены примеры путевых выражений на языке WebSQL:

a) -> | =>: путь единичной длины, локальный или глобальный;
b) ->*: локальный путь произвольной длины;
c) =>->*: то же, что и выше, но на других серверах;
d) (-> |=>)*: достижимая часть веба. ♦

Помимо путевых выражений в запросах, WebSQL позволяет уточнять область поиска во фразе **FROM**:

```
FROM Relation SUCH THAT domain-condition
```

где `domain-condition` может быть путевым выражением или определять поиск по тексту, если используется ключевое слово `MENTIONS`, или включать условие равенства атрибута (упоминаемого во фразе **SELECT**) веб-объекту. Разумеется, каждому упомянутому в запросе отношению может быть сопоставлена переменная, как в стандартном SQL. Приведенные ниже примеры запросов демонстрируют возможности WebSQL.

Пример 12.8. Ниже приведены примеры применения WebSQL:

a) в первом примере мы просто ищем все документы, содержащие слово «hypertext». При этом демонстрируется применение ключевого слова **MENTIONS** для уточнения области поиска:

```
SELECT D.URL, D.TITLE
FROM   DOCUMENT D
       SUCH THAT D MENTIONS "hypertext"
WHERE  D.TYPE = "text/html"
```

b) во втором примере демонстрируется два способа задания области поиска, а также поиск с учетом ссылок. Требуется найти все ссылки на апплеты в документах на тему «Java»:

```
SELECT A.LABEL, A.HREF
FROM   DOCUMENT D SUCH THAT D MENTIONS "Java"
       ANCHOR A SUCH THAT BASE=X
WHERE  A.LABEL = "applet"
```

c) в третьем примере демонстрируется использование различных типов ссылок. Требуется найти документы, в названии которых встречается строка «database», достижимые с домашней страницы цифровой библиотеки ACM по пути длиной не более 2, содержащему только локальные ссылки:

```
SELECT D.URL, D.TITLE
FROM   DOCUMENT D SUCH THAT "http://www.acm.org/dl"=|->|->-> D
WHERE  D.TITLE CONTAINS "database"
```

d) и в последнем примере демонстрируется одновременное задание условий поиска по содержимому и по структуре. Требуется найти все

документы с упоминанием «Computer Science», а также все документы, связанные с ними путями длины не более 2, содержащими только локальные ссылки:

```
SELECT D1.URL, D1.TITLE, D2.URL, D2.TITLE
FROM   DOCUMENT D1 SUCH THAT D1 MENTIONS "Computer Science",
       DOCUMENT D2 SUCH THAT D1=|->|->-> D2
```

♦

С помощью WebSQL можно опрашивать веб-данные на основе ссылок и текстового содержимого документов, но нельзя предъявлять запросы к структуре документа. Это следствие модели данных, в которой веб рассматривается как набор атомарных объектов.

Языки второго поколения, в т. ч. WebOQL, устраняют этот недостаток, моделируя веб как граф объектов, наделенных структурой. В каком-то смысле они сочетают некоторые возможности слабо структурированных подходов со средствами языков первого поколения.

Главная структура данных WebOQL называется *гипердеревом*, это упорядоченное дерево с двумя типами помеченных ребер: внутренними и внешними. *Внутреннее ребро* представляет внутреннюю структуру веб-документа, а *внешнее* – ссылку между объектами (т. е. гиперссылку). Каждое ребро помечено записью, включающей ряд атрибутов (полей). Для внешнего ребра эта запись должна содержать атрибут URL, кроме того, внешнее ребро должно оканчиваться листовым узлом гипердерева (не имеющим потомков).

Пример 12.9. Вернемся к примеру 12.2 и предположим, что моделируются не документы в библиографии, а документы, относящиеся к управлению данными в вебе. Возможное (частичное) гипердерево для этого примера показано на рис. 12.7. Мы внесли одно изменение, чтобы можно было предъявлять некоторые из показанных ниже запросов: добавили в каждый документ реферат.

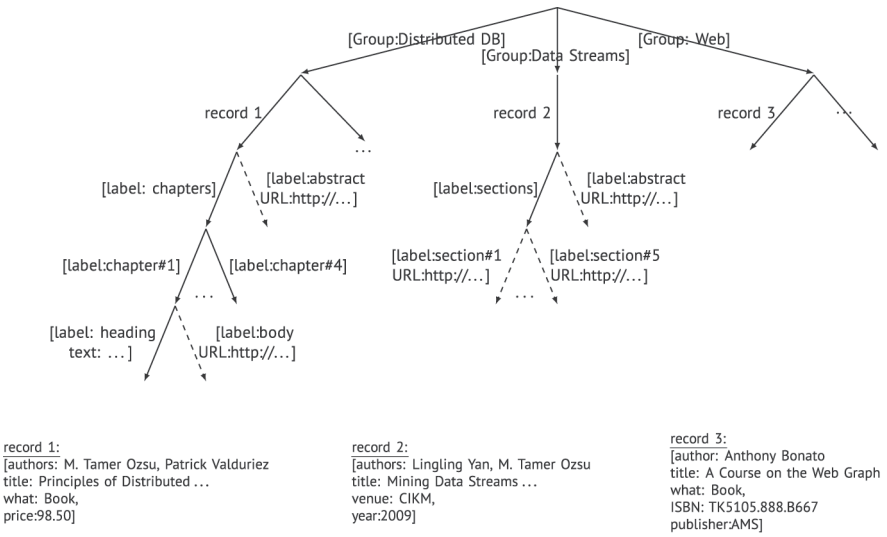


Рис. 12.7 ❖ Пример гипердерева

На рис. 12.7 документы сначала группируются по темам, как видно из записей, присоединенных к ребрам, исходящим из корневой вершины. В этом представлении внутренние ссылки изображаются сплошными линиями, а внешние – штриховыми. Напомним, что в OEM (рис. 12.5) ребра представляют как атрибуты (например, автора), так и структуру документа (например, главу). В модели WebOQL атрибуты хранятся в записях, ассоциированных с ребрами, а структуру документа представляют внутренние ребра. ♦

На базе этой модели WebOQL определяет операторы над деревьми.

Штрих: возвращает первое поддерево своего аргумента (обозначается ′).

Заглядывание: извлекает поле из записи, которая помечает первое ребро, исходящее из документа. Например, если *x* указывает на корень поддерева, достижимого по ребру «Groups = Distributed DB», то *x*.authors выделяет поле «M. Tamer Ozsu, Patrick Valduriez».

Конструирование: строит дерево с помеченными ребрами с записью, образованной аргументами (обозначается []).

Пример 12.10. Предположим, что дерево на рис. 12.8a является результатом запроса (назовем его Q1). Тогда выражение [label: «Papers by Ozsu» / Q1] строит дерево, показанное на рис. 12.8b. ♦

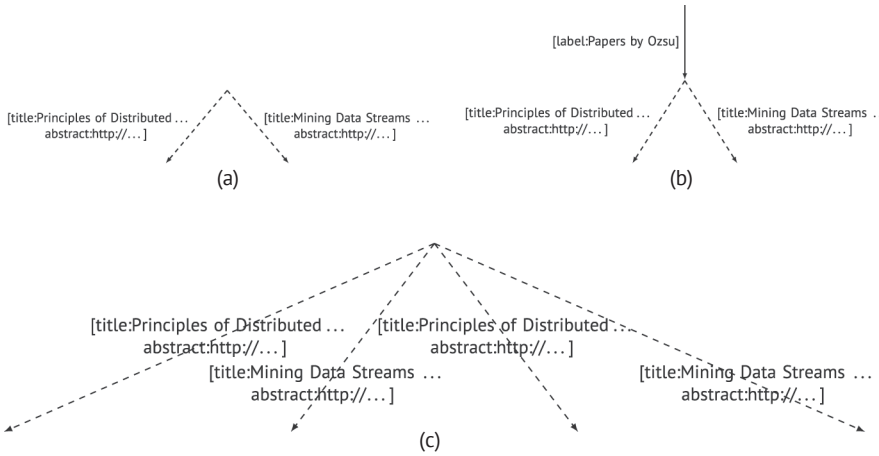


Рис. 12.8 ❖ Примеры операторов конструирования и конкатенации

Конкатенация: объединяет два дерева (обозначается +).

Пример 12.11. Снова предположим, что дерево на рис. 12.8a является результатом запроса Q1, тогда Q1+Q2 порождает дерево на рис. 12.8c. ♦

Голова: возвращает первое простое дерево дерева (обозначается &). Простым деревом дерева *t* называется дерево, состоящее из одного ребра, за которым следует (возможно, пустое) дерево, растущее из концевой вершины этого ребра.

Хвост: возвращает все деревья дерева, кроме первого простого (обозначается !).

Дополнительно в WebOQL имеется оператор сопоставления с образцом (обозначается ~), левым операндом которого является строка, а правым – образец. Поскольку язык поддерживает только один тип данных – строку, этот оператор важен.

WebOQL – функциональный язык, поэтому путем комбинирования этих операторов можно создавать сложные запросы. Кроме того, все операторы можно погружать в обычные запросы в духе SQL (или OQL), как показано в следующем примере.

Пример 12.12. Обозначим `dbDocuments` документы в базе данных, изображенной на рис. 12.7. Тогда следующий запрос находит названия и рефераты всех документов с автором «Ozsu» и возвращает результат, показанный на рис. 12.8а.

```
SELECT y.title, y′.URL
FROM   x IN dbDocuments, y IN x′
WHERE y.authors ~ "Ozsu"
```

Этот запрос интерпретируется следующим образом. Переменная `x` пробегает все простые деревья `dbDocuments`, и для каждого значения `x` переменная `y` перебирает простые деревья единственного поддерева `x`. Запрос заглядывает в запись ребра, и если поле `authors` сопоставляется со строкой «Ozsu» (для сравнения используется оператор ~), то конструируется дерево, метка которого содержит атрибут `title` из записи, на которую указывает `y`, и атрибут `URL` поддерева. ♦

В языках веб-запросов, обсуждаемых в этом разделе, принята более развитая модель данных, чем в слабо структурированных подходах. Эта модель улавливает как структуру документа, так и связи между разными документами. Кроме того, языки учитывают разную семантику ребер. Кроме того, как мы видели на примерах WebOQL, результатом запроса может быть новая структура. Однако для формулировки запросов все равно необходимы знания о структуре графа.

12.4. Вопросно-ответные системы

В этом разделе мы обсудим интересный и необычный (с точки зрения баз данных) подход к доступу к веб-данным: вопросно-ответные (QA) системы. Такие системы принимают вопросы на естественном языке, которые затем анализируются с целью определить, что имел в виду пользователь. После этого производится поиск для получения ответа.

Вопросно-ответные системы развивались в контексте систем информационного поиска, цель которых – найти ответ на поставленный вопрос в четко определенном корпусе документов. Их часто называют системами *с замкнутой областью*. Они расширяют возможность поиска по ключевым словам

в двух важнейших направлениях. Во-первых, пользователю разрешается задавать сложные вопросы на естественном языке, которые было бы трудно сформулировать в виде простого запроса по ключевым словам. В контексте веб-запросов это означает, что человек, задающий вопрос, не обязан знать, как именно организованы данные. Затем применяются сложные методы обработки естественного языка (ОЕЯ, англ. NLP) с целью понять смысл вопроса. Во-вторых, в результате поиска в корпусе документов возвращается явный ответ, а не ссылки на документы, релевантные запросу. Это не означает, что возвращаются точные ответы, как в традиционных СУБД, но может быть возвращен ранжированный список явных ответов, а не набор веб-страниц. Например, в ответ на запрос по ключевым словам «президент США» поисковая система вернула бы результат, показанный (частично) на рис. 12.9. Ожида-

Рис. 12.9 ❖ Пример поиска по ключевым словам

ется, что пользователь найдет ответ на страницах, URL и краткие выдержки из которых (цитаты) представлены на странице результатов. С другой стороны, в ответ на вопрос на естественном языке «Кто является президентом США?» система могла бы вернуть ранжированный список имен президентов (точный вид ответа зависит от системы).

Вопросно-ответные системы были обобщены для работы в вебе. В таких системах веб рассматривается как корпус документов (поэтому они называются системами *с открытой областью*). Доступ к источникам веб-данных осуществляется с помощью оберток, разработанных для получения ответов на запросы. Создан целый ряд вопросно-ответных систем с разными целями и функциональностью, например Mulder, WebQA, Start и Tritus. Существуют также коммерческие системы с различными возможностями (например, Wolfram Alpha http://www.wolframalpha.com/).

Мы опишем общую функциональность этих систем, ссылаясь на эталонную архитектуру на рис. 12.10. Этап предобработки, существующий не во всех системах, – это автономный процесс выделения и дополнения правил, используемых в системе. Во многих случаях он заключается в анализе документов, полученных из веба или возвращенных в ответ на ранее заданные вопросы. Цель такого анализа – определить наиболее эффективную структуру запроса, в которую следует преобразовать вопрос пользователя. Например, в Tritus применяется подход на основе машинного обучения с использованием в качестве обучающего набора часто задаваемых вопросов и правильных ответов на них. Система пытается угадать структуру ответа, для чего анализирует вопрос и ищет ответ на него в обучающем наборе. Процесс состоит из трех этапов. На первом этапе в результате анализа выделяется *вопросительная фраза* (например, в вопросе «What is a hard disk?» вопросительной фразой является «What is a»). Она служит для классификации вопроса. На втором

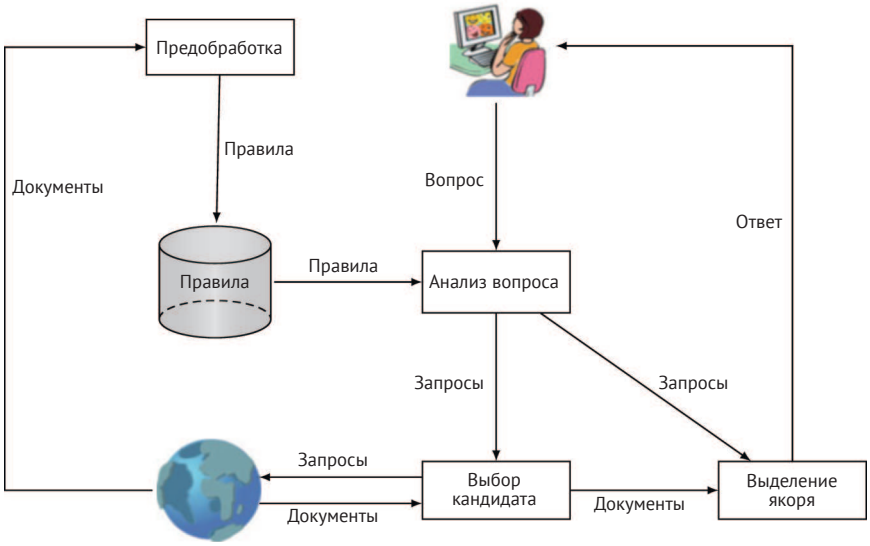


Рис. 12.10 ❖ Общая архитектура QA-систем

этапе анализируются пары вопрос-ответ в обучающих данных и генерируются *преобразования-кандидаты* для каждой вопросительной фразы. Например, для вопросительной фразы «What is a» (что такое) генерируются обороты «refers to» (относится к), «stands for» (означает) и т. д. На третьем этапе каждое преобразование-кандидат применяется к вопросам из обучающего набора и получившиеся запросы отправляются различным поисковым системам. Вычисляется сходство возвращенных результатов с правильными ответами в обучающем наборе, и на этой основе производится ранжирование преобразований-кандидатов. Ранжированные правила преобразования сохраняются для последующего использования при обработке реальных вопросов.

Заданный пользователем вопрос на естественном языке сначала проходит процедуру анализа. Ее цель – понять смысл вопроса. Большинство систем пытаются угадать тип вопроса, чтобы классифицировать его, а затем использовать результат при трансляции вопроса на язык запросов и при выделении ответа. Если предобработка проводилась, то в качестве подмоги используются созданные правила преобразования. Хотя общие цели всегда одинаковы, решения, предлагаемые в разных системах, значительно различаются в зависимости от развитости применяемых методов ОЕЯ (этот этап, как правило, целиком завязан на ОЕЯ). Например, анализ вопроса в системе Mulder состоит их трех стадий: синтаксический разбор вопроса, классификация вопроса и генерирование запроса. На первой стадии генерируется дерево разбора, которое используется при генерировании запроса и выделении ответа. На стадии классификации вопросу сопоставляется один из нескольких классов, например: *именной* для имен существительных, *числовой* – для чисел и *временной* – для дат. Такая классификация производится в большинстве QA-систем, потому что упрощает выделение ответа. Наконец, на стадии генерирования запроса созданное ранее дерево разбора применяется для конструирования одного или нескольких запросов, которые можно выполнить для получения ответов на вопрос. В Mulder на этой стадии используется четыре метода:

- замена глагола. Главный и вспомогательный глаголы заменяются спрягаемым глаголом (например, «When did Nixon visit China?» преобразуется в «Nixon visited China»);
- расширение запроса. Прилагательное в вопросительной фразе заменяется существительным (например, «How tall is Mt. Everest?» преобразуется в «The height of Everest is»);
- образование именной группы. Некоторые именные группы заключаются в кавычки, чтобы на следующей стадии передать их поисковой системе вместе;
- трансформация. Структура вопроса преобразуется в структуру ожидаемого типа ответа («Who was the first American in space?» преобразуется в «The first American in space was»).

Mulder – пример системы, в которой используются развитые средства ОЕЯ для анализа вопроса. На другом конце спектра находится система WebQA, которая относится к разбору вопроса не так серьезно.

После того как вопрос проанализирован и сгенерировано один или несколько запросов, наступает черед следующего шага – генерирования потен-

циальных ответов. Запросы, сгенерированные на стадии анализа вопроса, используются для поиска релевантных документов по ключевым словам. Многие системы на этом шаге просто обращаются к универсальным поисковым системам, но есть и такие, что прибегают к дополнительным источникам информации, доступным в вебе. Например, система ЦРУ World Factbook (https://www.cia.gov/library/publications/theworld-factbook/) – очень популярный источник надежной фактологической информации о странах. Аналогично надежные сведения о погоде можно получить из таких источников метеорологической информации, как Network (http://www.theweathernetwork.com/) или Weather Underground (http://www.wunderground.com/). Эти дополнительные источники иногда могут давать более качественные ответы, и разные системы пользуются ими в разной степени. Поскольку ответы на различные запросы лучше искать в разных источниках (а иногда даже в разных поисковых системах), важный аспект этой стадии обработки – выбор подходящих систем или источников данных. Наивное предъявление запросов всем поисковым системам или источникам данных – не лучшее решение, потому что в вебе эти операции обходятся весьма дорого. Обычно для выбора источников используется информация о категории вместе с ранжированными списками поисковых систем и источников для разных категорий. Для каждой поисковой системы или источника данных необходимо написать обертку, которая преобразует запрос в формат этой системы (источника), а полученные результаты – в единый формат для последующего анализа.

В ответ на запрос поисковые системы возвращают ссылки на документы вместе с краткими выдержками, а другие источники данных могут возвращать результаты в самых разных форматах. Полученные результаты преобразуются в «записи» единого формата. Из этих записей нужно извлечь прямые ответы – в этом и состоит стадия выделения ответов. Для сопоставления ключевых слов с записями (или их частями) применяются различные методы обработки текста. Затем результаты необходимо ранжировать, применяя различные приемы, заимствованные из информационного поиска (например, частота слов, обратная документная частота). В этом процессе используется информация о категории, сгенерированная на стадии анализа вопроса. У разных систем разное представление о том, что такое правильный ответ. Одни возвращают ранжированный список прямых ответов (например, на вопрос «Кто изобрел телефон» дают ответ «Александр Грейам Белл», или «Грейам Белл», или «Белл», или сразу три в определенном порядке[1]), тогда как другие возвращают части записей, содержащих встречающиеся в запросе слова (резюме релевантной части документа) в ранжированном порядке.

Вопросно-ответные системы сильно отличаются от других подходов к опросу веба, рассмотренных в предыдущих разделах. Они более гибкие, поскольку позволяют задавать вопросы, ничего не зная об организации вебданных. С другой стороны, они ограничены особенностями естественного языка и трудностями его обработки.

---

[1] Насчет того, кто изобрел телефон, есть разные мнения и несколько претендентов на почетное звание изобретателя. Мы остановились в этом примере на Белле, потому что он первым запатентовал телефон.

12.5. Поиск и опрос скрытого веба

В настоящее время большинство универсальных систем работают только с публично индексируемым вебом (ПИВ), хотя значительная часть ценных данных хранится на скрытых складах – в реляционных базах данных, в виде документов и во многих других формах. Наблюдается тенденция к отысканию путей поиска также в скрытом вебе, и тому есть две причины. Первая – размер: размер скрытого веба (в терминах генерируемых HTML-страниц) намного превышает размер ПИВ, поэтому вероятность найти ответы на запросы пользователей будет гораздо выше, если включить его в пространство поиска. Вторая – качество данных: данные, хранящиеся в скрытом вебе, обычно намного более высокого качества, чем те, что можно найти на открытых для всеобщего доступа страницах, поскольку их тщательно проверяют. Если бы к ним был доступ, то качество ответов тоже улучшилось бы.

Однако поиск в скрытом вебе наталкивается на многочисленные трудности. Перечислим самые важные из них.

1. Обычного робота нельзя использовать для поиска в скрытом вебе, поскольку нет ни HTML-страниц, ни ссылок, которые можно было бы посетить.
2. Как правило, к данным в скрытых базах можно получить доступ только через поисковый или иной специальный интерфейс, а значит, необходим доступ к этому интерфейсу.
3. В большинстве (если не во всех) случаев структура базы данных неизвестна, и поставщики данных неохотно предоставляют информацию о своих данных, которая могла бы помочь в процессе поиска (быть может, из-за затрат на сбор и обслуживание этой информации). Приходится работать через интерфейсы, предлагаемые источниками данных.

Далее в этой главе мы обсудим эти проблемы и некоторые предлагаемые решения.

12.5.1. Обход скрытого веба

Один из подходов к поиску в скрытом вебе – попробовать организовать обход так же, как для ПИВ. Мы уже говорили, что единственный способ работы со скрытой базой данных – воспользоваться ее поисковым интерфейсом. Робот для скрытого веба должен уметь решать две задачи: (a) отправлять запросы через поисковый интерфейс базы данных и (b) анализировать возвращенные страницы результатов и выделять из них релевантную информацию.

12.5.1.1. Запрос через поисковый интерфейс

Мы можем проанализировать поисковый интерфейс и построить его внутреннее представление. В этом представлении должны быть описаны поля интерфейса, их типы (например, текстовое поле, список, флажок и т. д.), их области значений (например, конкретные значения, как в случае списка, или произвольные текстовые строки, как в случае текстовых полей), а также

метки, ассоциированные с полями. Для выделения меток необходим полный анализ HTML-кода страницы.

Затем это представление сопоставляется с базой данных, созданной специально под задачу. Для сопоставления используются метки полей. Если метки совпадают, то в базу помещаются возможные значения поля. Этот процесс повторяется для всех возможных значений всех полей в поисковой форме, форма отправляется серверу со всеми возможными комбинациями значений, и полученные результаты анализируются.

Другой подход – воспользоваться технологией агентов. В этом случае разрабатываются *агенты скрытого веба*, которые взаимодействуют с поисковыми формами и извлекают страницы результатов. Процесс состоит из трех шагов: (a) найти формы, (b) научиться заполнять формы и (c) выявить и получить страницы результатов.

Для выполнения первого шага мы начинаем с некоторого URL (точки входа), проходим по ссылкам и применяем эвристики для определения HTML-страниц, содержащих формы, исключая те, что содержат поля пароля (вход в систему, регистрация, страница покупки). Для заполнения формы нужно выявить метки и связать их с полями формы. Для этого применяются эвристические знания о вероятном расположении метки относительно поля (слева от него или над ним). Зная метки, агент определяет предметную область, к которой относится форма, и заполняет поля значениями из этой предметной области в соответствии с метками (значения хранятся в доступном агенту репозитории).

12.5.1.2. Анализ страниц результатов

После того как форма отправлена, возвращенную страницу необходимо проанализировать и понять, содержит ли она данные или форму для уточнения поиска. Для этого можно сравнить значения на этой странице со значениями из репозитория агента. Если это страница данных, то обходится как она сама, так и все страницы, на которые она указывает (в особенности страницы с дополнительными результатами). Это продолжается до тех пор, пока не иссякнут все страницы, принадлежащие одному домену.

Однако возвращенные страницы обычно содержат много лишней информации, помимо собственно результатов, т. к. в большинстве случаев они создаются по шаблону, в котором присутствует много текста рекламного характера. Для распознавания шаблонов страницы анализируется текстовое содержимое и структура соседних тегов, чтобы выделить относящиеся к запросу данные. Веб-страница представлена в виде последовательности текстовых сегментов, т. е. участков текста, заключенных между двумя тегами. Механизм распознавания шаблонов устроен следующим образом:

1) текстовые сегменты анализируются на предмет текстового содержимого и соседних тегов;
2) начальный шаблон идентифицируется по первым двум выборочным документам;
3) шаблон генерируется, если в обоих документах найдены соответствующие текстовые сегменты и соседние с ними теги;

4) последующие документы сравниваются со сгенерированным шаблоном. Из каждого документа извлекаются для последующей обработки текстовые сегменты, отсутствующие в шаблоне;
5) если не найдено совпадений с существующим шаблоном, то содержимое документа извлекается для генерирования будущих шаблонов.

12.5.2. Метапоиск

Другой подход к опросу скрытого веба называется метапоиском. Получив запрос пользователя, метапоисковик выполняет следующие действия:

1) **выбор базы данных.** Выбирается база (или базы) данных, наиболее релевантная запросу. Для этого необходимо собрать информацию о каждой базе данных. Эта информация называется *аннотацией содержимого* и представляет собой статистическую информацию, которая обычно включает *документные частоты* слов, встречающихся в базе данных;
2) **трансляция запроса.** Запрос транслируется в форму, подходящую для каждой базы данных (например, путем заполнения некоторых полей в поисковом интерфейсе базы данных);
3) **объединение результатов.** Данные собираются из разных баз, объединяются (скорее всего, также упорядочиваются) и возвращаются пользователю.

Ниже мы подробно обсудим важные этапы метапоиска.

12.5.2.1. Выделение резюме содержимого

Первый шаг метапоиска – вычислить резюме содержимого. В большинстве случаев поставщики данных не горят желанием заниматься подготовкой такой информации. Поэтому метапоисковик выделяет ее самостоятельно.

Один из возможных подходов – сделать выборку документов из базы данных D и для встречающегося в ней слова w вычислить частоту $SampleDF(w)$. Делается это так:

1) начать с пустого резюме содержимого, в котором для каждого слова w частота $SampleDF(w) = 0$. Кроме того, взять общий (не специализированный для D) словарь;
2) выбрать слово и отправить его в качестве запроса к базе данных D;
3) из возвращенных документов отобрать первые k;
4) если количество возвращенных документов превышает заранее заданный порог, остановиться. В противном случае вернуться к шагу 2.

У этого алгоритма есть две основные версии, различающиеся тем, как выполняется шаг 2. В первом случае из словаря выбирается случайное слово. Во втором слово для следующего запроса выбирается из тех, что уже были обнаружены в процессе выборки. Первый подход позволяет построить более точный профиль, но обходится дороже.

Альтернатива – воспользоваться методом сфокусированного апробирования, который позволяет построить иерархическую классификацию баз

данных. Идея в том, чтобы заранее разнести обучающие документы по нескольким категориям, а затем выделять из документов термы и использовать их в качестве апробирующих запросов. Запросы из одного слова помогают определить *фактические* документные частоты этих слов, при этом для других слов, встречающихся в более длинных пробах, вычисляются только *выборочные* документные частоты. Они используются в качестве оценки фактических документных частот слов.

Еще один подход состоит в том, чтобы начать со случайного выбора терма из самого поискового интерфейса в предположении, что этот терм с большой вероятностью связан с содержимым базы данных. Терм запрашивается у базы данных, и извлекаются первые k документов. Затем из множества термов, встретившихся в извлеченных документах, случайным образом выбирается следующий терм. Этот процесс повторяется, пока не будет получено заранее заданное число документов, после чего по ним вычисляется статистика.

12.5.2.2. Категоризация баз данных

Хороший подход, помогающий в процессе выбора баз данных, – разнести базы данных по нескольким категориям (как, например, в каталоге Yahoo). Категоризация помогает найти базу данных, отвечающую запросу пользователя, и сделать большинство возвращенных результатов релевантными запросу.

Если для генерирования резюме содержимого применяется метод сфокусированного апробирования, то его же можно использовать, чтобы апробировать каждую базу данных запросами из некоторой категории и подсчитать количество соответствий. Если количество соответствий превышает порог, то считается, что база данных принадлежит этой категории.

Выбор базы данных

Выбор базы данных – самая важная задача в процессе метапоиска, поскольку от нее критически зависят эффективность и качество обработки запросов к нескольким базам. Алгоритм выбора базы данных пытается найти наилучший набор баз, которым стоит отправить запрос, на основе информации об их содержимом. Обычно эта информация включает количество документов, содержащих каждое слово (эта величина называется документной частотой), и другую простую статистику, например количество документов, хранящихся в базе. Располагая этой сводной информацией, алгоритм выбора оценивает релевантность каждой базы данных запросу (например, с точки зрения ожидаемого количества возвращенных базой результатов).

GlOSS – простой алгоритм выбора баз данных, в котором предполагается, что входящие в запрос слова независимо распределены по документам базы, и на основе этой гипотезы оценивается количество документов, отвечающих запросу. GlOSS – представитель большого семейства алгоритмов выбора баз данных, опирающихся на резюме содержимого. Такие алгоритмы ожидают, что резюме точны и актуальны.

Рассмотренный выше алгоритм сфокусированного апробирования пользуется категоризацией баз данных и резюме содержимого для выбора баз. Он

состоит из двух шагов: (1) распространить резюме содержимого баз данных на категории иерархической схемы классификации и (2) использовать резюме содержимого категорий и баз данных, чтобы выбрать базы, сконцентрировавшись на самых релевантных частях тематической иерархии. Это позволяет дать ответы, более релевантные запросу пользователя, потому что для их получения будут использоваться только базы, принадлежащие той же категории, что и сам запрос.

После того как релевантные базы выбраны, каждой из них отправляется запрос, и возвращенные результаты объединяются и предъявляются пользователю.

12.6. Интеграция веб-данных

В главе 7 мы обсуждали интеграцию баз данных, имеющих четко определенные схемы. Рассмотренные там методы больше подходят для корпоративных данных. Если же мы хотим предоставить интегрированный доступ к источникам веб-данных, то задача усложняется – начинают играть роль все характеристики «больших данных». В частности, данные могут не иметь схемы, а если и имеют, то источники данных настолько различаются, что схемы не имеют ничего общего, и их сопоставление оказывается весьма трудной проблемой. Кроме того, объем данных и даже количество источников данных намного больше, чем в корпоративном окружении, поэтому ручная проверка невозможна в принципе. Качество веб-данных также вызывает гораздо больше сомнений, чем в случае данных предприятия, поэтому становятся особенно важны процедуры очистки данных.

Разумным подходом к интеграции веб-данных является *интеграция с оплатой по факту*, при которой предварительные инвестиции в интеграцию данных значительно сокращаются, поскольку исключаются некоторые этапы, упомянутые в главе 7. Вместо этого владельцам данных предоставляется базовая инфраструктура, упрощающая объединение их наборов данных в федерацию. В одном из подобных предложений – *пространствах данных* – предполагается, что для начала должна существовать платформа облегченной интеграции, возможно, с рудиментарными возможностями доступа (например, поиск по ключевым словам), а также способы со временем повысить ценность интеграции за счет разработки инструментов для более продвинутого использования. *Озера данных*, которые завоевывают все больше сторонников и которые мы обсуждали в главе 10, по-видимому, являются более развитым вариантом пространств данных.

В этом разделе мы остановимся на некоторых подходах, предложенных для преодоления описанных выше трудностей. В частности, рассмотрим веб-таблицы и фьюжн-таблицы (раздел 12.6.1) в качестве сравнительно дешевого подхода к интеграции табличных данных. Затем мы поговорим о семантическом вебе и подходе Linked Open Data (LOD) к интеграции веб-данных (раздел 12.6.2.3). И наконец, в разделе 12.6.3 обсудим вопросы очистки данных и использование методов машинного обучения для интеграции и очистки в масштабе веба.

12.6.1. Веб-таблицы и фьюжн-таблицы

Два популярных подхода к облегченной интеграции веб-данных – веб-порталы и *гибридные веб-приложения* (mashup), которые агрегируют веб-данные и другую информацию по конкретным темам, например путешествия, бронирование мест в гостиницах и т. д. Отличаются они используемыми технологиями, но для нашего обсуждения это не важно. Это примеры «вертикально интегрированных» систем, когда каждый гибрид или портал ориентирован на одну предметную область.

Первый вопрос, возникающий при разработке гибридного приложения: как найти релевантные веб-данные? Проект веб-таблиц – одна из первых попыток найти в вебе данные, имеющие структуру реляционной таблицы, и предоставить доступ к этим таблицам (так называемым «таблицам как в базе данных»). Проект распространялся только на открытый веб, поскольку выявление таблиц в глубинном вебе – задача гораздо более трудная. Даже в открытом вебе найти таблицы как в базе данных нелегко, потому что привычной реляционной структуры (например, имен атрибутов) может и не быть. В веб-таблицах используется классификатор, который умеет определять HTML-таблицу как реляционную или нереляционную. Предоставляются также инструменты для выделения схемы и сбора статистики о схемах, которой можно воспользоваться при поиске в этих таблицах. Предлагаются средства соединения обнаруженных таблиц для более развитой навигации. Веб-таблицы можно рассматривать как метод извлечения и опроса веб-данных, но они также играют роль каркаса виртуальной интеграции для веб-данных, наделенных информацией о глобальной схеме.

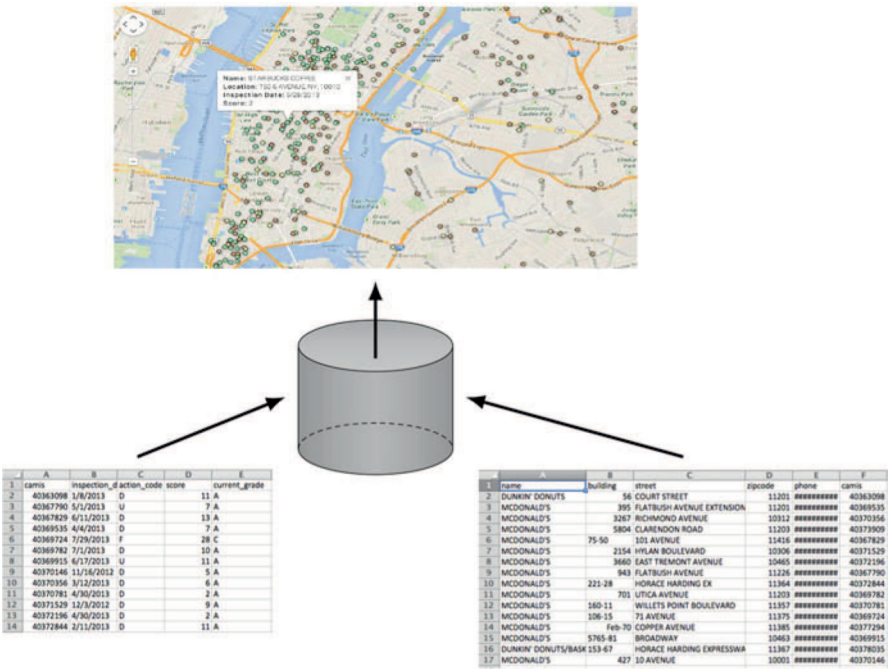
Фьюжн-таблицы – это проект Google, который заходит дальше и позволяет пользователям загружать собственные таблицы (в различных форматах) в дополнение к обнаруженным веб-таблицам. Инфраструктура фьюжн-таблиц способна автоматически распознавать атрибут, по которому соединяются таблицы, и предлагает возможности интеграции. На рис. 12.11 изображены два набора данных от разных владельцев: один описывает заведения питания, а другой – оценки, присвоенные этим заведениям инспекторами. Система должна установить, что эти два набора данных можно соединить по общему атрибуту, и предложить интегрированный доступ. В этом случае обе таблицы были предоставлены пользователями, но в других одна или обе таблицы могут быть найдены в вебе с помощью методов, разработанных для проекта веб-таблиц.

12.6.2. Семантический веб и проект Linked Open Data

Веб можно рассматривать как огромный репозиторий данных, *допускающих машинную обработку*. Идея семантического веба заключается в том, чтобы преобразовать эти данные в *машинно воспринимаемую* форму, интегрировав структурированные и неструктурированные данные в вебе и снабдив

их семантической разметкой. Оригинальное видение семантического веба включало три компоненты:

- разметить веб-данные, присоединив к ним метаданные в виде аннотаций;
- использовать онтологии, чтобы различные наборы данных можно было понять;
- использовать основанные на математической логике технологии для доступа к метаданным и онтологиям.



	A	B	C	D	E
1	camis	inspection_d	action_code	score	current_grade
2	40363098	1/8/2013	D	11	A
3	40367790	5/1/2013	U	7	A
4	40367829	6/11/2013	D	13	A
5	40369535	4/4/2013	D	7	A
6	40369724	7/29/2013	F	28	C
7	40369782	7/1/2013	D	10	A
8	40369915	6/17/2013	U	11	A
9	40370146	11/16/2012	D	5	A
10	40370356	3/12/2013	D	6	A
11	40370781	4/30/2013	D	2	A
12	40371529	12/3/2012	D	9	A
13	40372196	4/30/2013	D	2	A
14	40372844	2/11/2013	D	11	A

	A	B	C	D	E	F
1	name	building	street	zipcode	phone	camis
2	DUNKIN' DONUTS	56	COURT STREET	11201	##########	40363098
3	MCDONALD'S	395	FLATBUSH AVENUE EXTENSION	11201	##########	40369535
4	MCDONALD'S	3267	RICHMOND AVENUE	10312	##########	40370356
5	MCDONALD'S	5804	CLARENDON ROAD	11203	##########	40373909
6	MCDONALD'S	75-50	101 AVENUE	11416	##########	40367829
7	MCDONALD'S	2154	HYLAN BOULEVARD	10306	##########	40371529
8	MCDONALD'S	3660	EAST TREMONT AVENUE	10465	##########	40372196
9	MCDONALD'S	943	FLATBUSH AVENUE	11226	##########	40367790
10	MCDONALD'S	221-28	HORACE HARDING EX	11364	##########	40372844
11	MCDONALD'S	701	UTICA AVENUE	11203	##########	40369782
12	MCDONALD'S	160-11	WILLETS POINT BOULEVARD	11357	##########	40370781
13	MCDONALD'S	106-15	71 AVENUE	11375	##########	40369724
14	MCDONALD'S	Feb-70	COPPER AVENUE	11385	##########	40377294
15	MCDONALD'S	5765-81	BROADWAY	10463	##########	40369915
16	DUNKIN' DONUTS/BASK	153-67	HORACE HARDING EXPRESSWA	11367	##########	40378035
17	MCDONALD'S	427	10 AVENUE	10001	##########	40370146

Рис. 12.11 ❖ Пример веб-таблиц и фьюжн-таблиц

Проект Linked Open Data (LOD) был инициирован в 2006 году как уточнение этого видения, при этом подчеркивалось значение связей между данными, являющимися частью семантического веба. Были сформулированы инструкции, описывающие, как следует публиковать данные в вебе, чтобы достичь целей семантического веба. Таким образом, семантический веб – это видение интеграции веб-данных, реализованное посредством LOD. Требования LOD к публикации (а значит, и интеграции) данных в вебе базируются на четырех принципах:

- все веб-ресурсы (данные) локально идентифицируются своими URI, которые играют роль имен;
- эти имена доступны по протоколу HTTP;
- информация о веб-ресурсах (сущностях) кодируется тройками RDF (Resource Description Framework). Иными словами, RDF – это модель данных в семантическом вебе (мы обсудим ее ниже);

- связи между наборами данных устанавливаются с помощью ссылок на данные, и издатели наборов данных должны задавать эти ссылки, чтобы можно было найти больше данных.

Стало быть, LOD порождает граф, вершинами которого являются веб-ресурсы, а ребрами – связи между ними. Упрощенная форма «LOD-графа» по состоянию на 2018 год показана на рис. 12.12. Каждая вершина представляет набор данных (не веб-ресурс), классифицированный согласно цвету (например, публикации, науки о жизни, социальные сети), а размер вершины представляет ее входящую степень. Тогда LOD состоял из 1234 наборов данных, связанных 16 136 ссылками[1]. Мы вернемся к проекту LOD и LOD-графу чуть ниже.

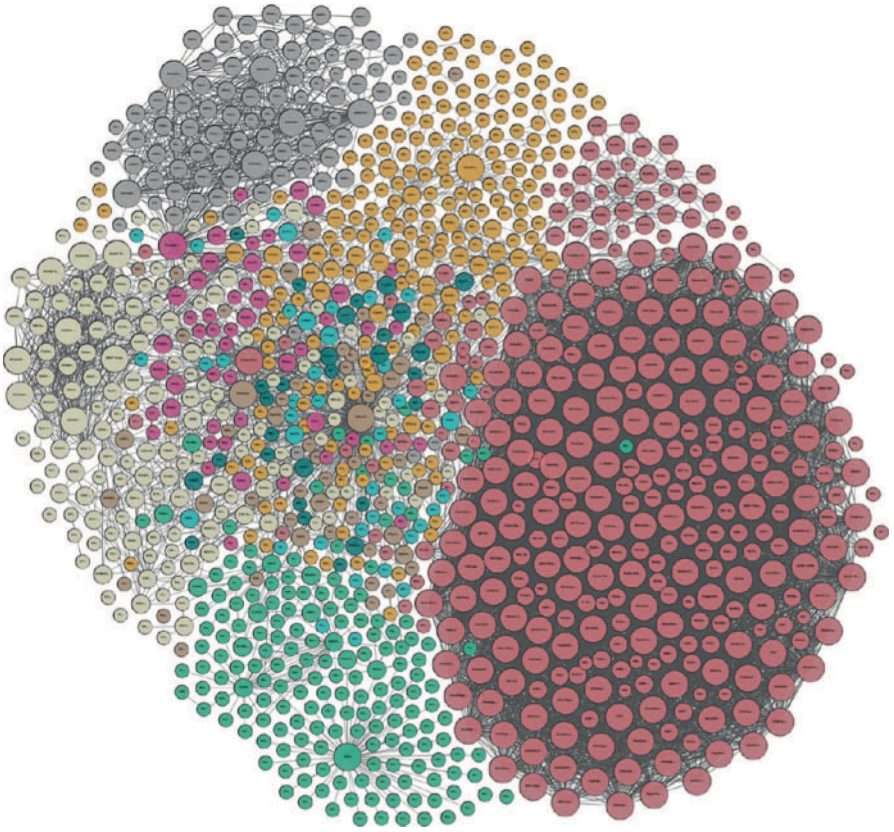


Рис. 12.12 ❖ LOD-граф по состоянию на 2018 год

На нижнем уровне XML дает язык для записи структурированных веб-документов и простого обмена такими документами (рис. 12.13). Над ним надстроен каркас RDF, который, как мы уже сказали, устанавливает модель

[1] Статистика взята с сайта https://lod-cloud.net, и там же следует искать более свежие данные.

данных. RDF-схема необязательна, но если она задана, то предоставляет необходимые примитивы. Онтологии дополняют RDF-схему более мощными конструкциями, позволяющими описывать связи между данными. Наконец, основанные на математической логике декларативные языки задания правил позволяют приложениям определять собственные правила.

Рис. 12.13 ❖ Технологии семантического веба. На основе работы [Antoniou and Plexousakis 2018]

Далее мы обсудим технологии, расположенные на нижних уровнях, потому что они составляют минимальный набор требований.

12.6.2.1. XML

Веб-документы кодировались в основном на языке HTML (HyperText Markup Language – язык гипертекстовой разметки). Веб-документ, написанный на HTML, состоит из *элементов HTML*, обрамленных тегами; мы обсуждали это в разделе 12.6.1, где также рассказали о подходах к выявлению структурированных данных в HTML-кодированных веб-документах и их интеграции. Но в контексте семантического веба предпочтительным средством кодирования и обмена документами является XML (Extensible Markup Language – расширяемый язык разметки), предложенный консорциумом World Wide Web Consortium (W3C).

XML-теги (или разметка) разбивают данные на части, называемые *элементами*, с целью наделить данные семантикой. Элементы могут быть вложенными, но не могут перекрываться. Вложенность элементов представляет иерархические отношения между ними. Так, на рис. 12.14 показано XML-представление (с небольшими изменениями) встречавшихся нам ранее библиографических данных.

Любой XML-документ можно представить в виде дерева, содержащего *корневой элемент*, в который вложено ноль или более подэлементов (или *дочерних элементов*), которые сами могут содержать подэлементы. У каждого элемента имеется ноль или более *атрибутов*, значения которых имеют атомарные типы. Сам элемент также может иметь значение, хотя это необязательно. Текстовое представление дерева индуцирует отношение полного порядка между элементами, который называется *документным порядком*; оно определяется порядком следования элементов в документе.

Например, корневым элементом на рис. 12.4 является `bib`, и у него есть три дочерних элемента: два элемента `book` и один `article`. У первого элемента `book`

имеется атрибут `year` со значением «1999», а также подэлементы (например, `title`). Значением элемента `title` является строка «Principles of Distributed Database Systems»).

```
<bib>
<book year = "1999">
<author> M. Tamer Ozsu </author>
<author> Patrick Valduriez </author>
<title> Principles of Distributed ... </title>
<chapters>
<chapter>
<heading> ... </heading>
<body> ... </body>
</chapter>
...
<chapter>
<heading> ... </heading>
<body> ... </body>
</chapter>
</chapters>
<price currency= "USD"> 98.50 </price>
</book>
<article year = "2009">
<author> M. Tamer Ozsu </author>
<author> Yingying Tao </author>
<title> Mining data streams ... </title>
<venue> "CIKM" </venue>
<sections>
<section> ... </section>
...
<section> ... </section>
</sections>
</article>
<book>
<author> Anthony Bonato </author>
<title> A Course on the Web Graph </title>
<ISBN> TK5105.888.B667 </ISBN>
<chapters>
<chapter>
<heading> ... </heading>
<body> ... </body>
</chapter>
<chapter>
<heading> ... </heading>
<body> ... </body>
</chapter>
<chapter>
<heading> ... </heading>
<body> ... </body>
</chapter>
</chapters>
<publisher> AMS </publisher>
</book>
</bib>
```

Рис. 12.14 ❖ Пример XML-документа

Стандартное определение XML-документа несколько сложнее: элементы могут иметь атрибуты ID и IDREF, описывающие ссылки между элементами в одном и том же или в разных документах. В таком случае представление документа становится графом общего вида. Но очень часто используют более простое древовидное представление, мы так и будем поступать в этом разделе и ниже определим его более точно[1].

XML-документ моделируется как упорядоченное дерево с помеченными узлами $T = (V, E)$, каждый узел которого $v \in V$ соответствует элементу или атрибуту и характеризуется:

- уникальным идентификатором $ID(v)$;
- свойством $kind(v)$, принимающим значения {элемент, атрибут, текст};
- меткой $label(v)$, символы которой берутся из некоторого алфавита;
- содержимым $content(v)$, которое пусто для нелистовых узлов и является строкой для листовых.

Ориентированное ребро $e = (u, v)$ включается в E тогда и только тогда, когда:

- $kind(u) = kind(v)$ = элемент и v является подэлементом u или
- $kind(u)$ = элемент $\wedge$ $kind(v)$ = атрибут и v является атрибутом u.

Определив дерево XML-документа, мы можем определить экземпляр модели данных XML как упорядоченную последовательность узлов этого дерева или атомарных значений. Для XML-документа может быть определена схема, но это необязательно, поскольку XML – самоописываемый формат. Если для коллекции XML-документов определена схема, то все документы этой коллекции должны быть согласованы со схемой; однако схема допускает вариации, т. е. не все элементы или атрибуты обязательно должны быть представлены в каждом документе. XML-схемы определяются в виде определения типа документа (Document Type Definition – DTD) или на языке XML-Schema. В этом разделе мы будем работать с более простым определением схемы, в котором используется определенная выше графовая структура XML-документов.

Графом XML-схемы называется пятерка $\langle \Sigma, \Psi, s, m, \rho \rangle$, где Σ – алфавит типов узлов XML-документа, ρ – тип корневого узла, $\Psi \subseteq \Sigma \times \Sigma$ – множество ребер, соединяющих типы узлов, $s : \Psi \to \{\text{ONCE, OPT, MULT}\}$ и $m : \Sigma \to \{\texttt{string}\}$. Семантика этого определения такова. Ребро $\psi = (\sigma_1, \sigma_2) \in \Sigma$ означает, что узел типа σ_1 может содержать узел типа σ_2. $s(\psi)$ обозначает кратность связи, представленной этим ребром. Если $s(\psi)$ = ONCE, то узел типа σ_1 должен содержать ровно один узел типа σ_2. Если $s(\psi)$ = OPT, то узел типа σ_1 может содержать, а может и не содержать узел типа σ_2. Если $s(\psi)$ = MULT, то узел типа σ_1 может содержать несколько узлов типа σ_2. $m(\sigma)$ обозначает область значений текстового содержимого узла типа σ, представленную в виде множества всех строк, которые могут встречаться внутри такого узла.

Имея определения модели XML-данных и экземпляров этой модели, мы можем определить языки запросов. Выражение на языке запросов к XML принимает экземпляр XML-данных и возвращает экземпляр XML-данных.

---

[1] Мы также опускаем узлы комментариев, пространств имен и команд обработки, присутствующие в общей модели.

Консорциумом W3C предложено два таких языка запросов: XPath и XQuery. Путевые выражения, с которыми мы познакомились выше, встречаются в обоих языках и, пожалуй, являются самым естественным способом опроса иерархических XML-данных. В языке XQuery определены и более мощные конструкции. Хотя XQuery был предметом интенсивных исследований и разработок в 2000-х годах, сейчас он используется редко. Он сложен, неудобен для формулирования запросов и с трудом поддается оптимизации системой. Во многих приложениях на смену XML и XQuery пришел язык JSON, который мы обсуждали в главе 11, хотя XML-представление (но не XQuery) остается важным для семантического веба.

12.6.2.2. RDF

Модель данных RDF надстроена над XML и является фундаментальной частью семантического веба (рис. 12.13). Первоначально она была предложена консорциумом W3C как компонента семантического веба, но теперь применяется в более широком контексте. Например, системы Yago и DBPedia автоматически извлекают факты из Википедии и сохраняют их в формате RDF, чтобы поддержать структурные запросы к Википедии. В биологии с помощью RDF кодируются эксперименты и их результаты, что облегчает обмен данными между учеными; это привело к появлению таких наборов RDF-данных, как Bio2RDF (bio2rdf.org) и Uniprot RDF (dev.isb-sib.ch/projects/uniprot-rdf). Что касается семантического веба, то в проекте LOD строится облако RDF-данных путем связывания большого числа наборов данных, о чем мы уже упоминали выше.

RDF моделирует каждый «fact» в виде множества троек ⟨субъект, свойство (или предикат)⟩. В соответствии с английским написанием этих слов – **s**ubject, **p**roperty (или **p**redicate), **o**bject – тройка обозначается ⟨*s*, *p*, *o*⟩, где *субъектом* является сущность, класс или пустой узел, *свойство*[1] обозначает один атрибут, ассоциированный с одной сущностью, а *объект* является сущностью, классом, пустым узлом или литеральным значением. В стандарте RDF сущность обозначается своим URI (унифицированным идентификатором ресурса), который ссылается на именованный *ресурс* в моделируемой среде. Напротив, пустые узлы ссылаются на анонимные ресурсы, не имеющие имени[2]. Таким образом, каждая тройка представляет именованную связь; тройки, в которых встречаются пустые узлы, просто означают, что «нечто с такой связью существует, хотя и не поименовано».

Сейчас будет уместно сказать несколько слов о следующем уровне стека технологий семантического веба (рис. 12.13) – RDF-схеме (RDFS). С помощью RDFS, которая также является стандартом W3C, мы можем аннотировать RDF-данные семантическими метаданными[3]. Задача аннотаций состоит пре-

---

[1] В литературе термины «свойство» и «предикат» употребляются как синонимы, но мы будем всюду использовать термин «свойство».

[2] Во многих исследовательских работах пустые узлы игнорируются. Если явно не оговорено противное, то и мы будем игнорировать их в этой книге.

[3] Такую же аннотацию можно создать с помощью онтологических языков, например OWL (также стандарт W3C), но мы не станем развивать эту тему.

жде всего в том, чтобы дать возможность рассуждать о RDF-данных (это называется *импликацией*), но иногда они также влияют на организацию данных, а метаданные могут использоваться для оптимизации семантического запроса. Мы проиллюстрируем фундаментальные концепции на простых примерах RDFS, которые позволяют определять *классы* и *иерархии классов*. В RDFS имеются встроенные определения классов – самые важные из них `rdfs:Class` и `rdfs:subClassOf` служат для определения класса и подкласса (еще одно, `rdfs:label`, используется в примерах запросов ниже). Чтобы указать, что некий ресурс является элементом класса, применяется специальное свойство `rdf:type`.

Пример 12.13. Для определения класса `Movies` и двух его подклассов `ActionMovies` и `Dramas` нужно написать:

```
Movies rdf:type rdfs:Class .
ActionMovies rdfs:subClassOf Movies .
Dramas rdfs:subClassOf Movies .
```
♦

Формально набор данных RDF можно определить следующим образом. Обозначим $\mathcal{U}$, $\mathcal{B}$, $\mathcal{L}$ множества всех URI, пустых узлов и литералов соответственно. Кортеж $(s, p, o) \in (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ называется *RDF-тройкой*. Множество RDF-троек образует *набор данных RDF*.

Пример 12.14. На рис. 12.15 показан пример набора данных RDF, в котором данные взяты из источников, определенных префиксами URI. ♦

RDF-данные можно следующим образом смоделировать в виде RDF-графа. *RDF-графом* называется шестерка $G = \langle V, L_V, f_V, E, L_E, f_E \rangle$, где:

1) $V = V_c \cup V_e \cup V_l$ – множество вершин, соответствующих всем субъектам и объектам в RDF-данных, а V_c, V_e и V_l – соответственно множества вершин классов, вершин сущностей и вершин литералов;
2) L_V – множество меток вершин;
3) *функция пометки вершин* $f_V : V \to L_V$ – биективная функция, сопоставляющая каждой вершине метку. Меткой вершины $u \in V_l$ является ее литеральное значение, а меткой вершины $u \in V_c \cup V_e$ – соответствующий ей URI;
4) $E = \{\overrightarrow{u_1, u_2}\}$ – множество ориентированных ребер, которые соединяют соответственные субъекты и объекты;
5) L_E – множество меток ребер;
6) *функция пометки ребер* $f_E : E \to L_E$ – биективная функция, сопоставляющая каждому ребру метку. Меткой ребра $e \in E$ является соответствующее ему свойство.

Ребро $\overrightarrow{u_1, u_2}$ называется ребром *свойства атрибута*, если $u_2 \in V_l$, в противном случае – ребром *связи*.

Заметим, что структура RDF-графа отличается от структуры графов свойств, которые мы обсуждали в главе 10. Напомним, что в графах свойств к вершинам и ребрам присоединены атрибуты, что позволяет задавать в запросах сложные основанные на значениях предикаты. В RDF-графах единственными атрибутами вершин и ребер являются метки. То, что в графе свойств было

бы атрибутами вершины, становится ребрами, метками которых являются имена атрибутов. Поэтому RDF-графы проще и более регулярны, но в общем случае имеют больше вершин и ребер.

Префиксы:
mdb=http://data.linkedmdb.org/resource/ geo=http://sws.geonames.org/
bm=http://wifo5-03.informatik.uni-mannheim.de/bookmashup/
exvo=http://lexvo.org/id/
wp=http://en.wikipedia.org/wiki/

Subject	Property	Object
mdb: film/2014	rdfs:label	“The Shining”
mdb:film/2014	movie:initial_release_date	“1980-05-23”’
mdb:film/2014	movie:director	mdb:director/8476
mdb:film/2014	movie:actor	mdb:actor/29704
mdb:film/2014	movie:actor	mdb: actor/30013
mdb:film/2014	movie:music_contributor	mdb: music_contributor/4110
mdb:film/2014	foaf:based_near	geo:2635167
mdb:film/2014	movie:relatedBook	bm:0743424425
mdb:film/2014	movie:language	lexvo:iso639-3/eng
mdb:director/8476	movie:director_name	“Stanley Kubrick”
mdb:film/2685	movie:director	mdb:director/8476
mdb:film/2685	rdfs:label	“A Clockwork Orange”
mdb:film/424	movie:director	mdb:director/8476
mdb:film/424	rdfs:label	“Spartacus”
mdb:actor/29704	movie:actor_name	“Jack Nicholson”
mdb:film/1267	movie:actor	mdb:actor/29704
mdb:film/1267	rdfs:label	“The Last Tycoon”
mdb:film/3418	movie:actor	mdb:actor/29704
mdb:film/3418	rdfs:label	“The Passenger”
geo:2635167	gn:name	“United Kingdom”
geo:2635167	gn:population	62348447
geo:2635167	gn:wikipediaArticle	wp:United_Kingdom
bm:books/0743424425	dc:creator	bm:persons/Stephen+King
bm:books/0743424425	rev:rating	4.7
bm:books/0743424425	scom:hasOffer	bm:offers/0743424425amazonOffer
lexvo:iso639-3/eng	rdfs:label	“English”
lexvo:iso639-3/eng	lvont:usedIn	lexvo:iso3166/CA
lexvo:iso639-3/eng	lvont:usesScript	lexvo:script/Latn

Рис. 12.15 ❖ Пример набора данных RDF.
Для идентификации источников данных используются префиксы

На рис. 12.16 приведен пример RDF-графа. Вершины, обозначенные прямоугольниками, – это вершины сущностей и классов, остальные – вершины литералов.

Стандартным языком W3C для RDF является SPARQL, который определяется следующим образом [Hartig 2012]. Обозначим $\mathcal{U}$, $\mathcal{B}$, $\mathcal{L}$, $\mathcal{V}$ множества всех URI, пустых узлов, литералов и переменных соответственно. Выражение языка SPARQL определяется рекурсивно.

1. *Тройка* $(\mathcal{U} \cup \mathcal{B} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{V})$ является выражением SPARQL.
2. (Факультативно) Если P – выражение SPARQL, then P *FILTER* R – тоже выражение SPARQL, где R – встроенное в SPARQL условие фильтрации.

3. (Факультативно) Если P_1 и P_2 – выражения SPARQL, то P_1 *AND|OPT|OR* P_2 – тоже выражения SPARQL.

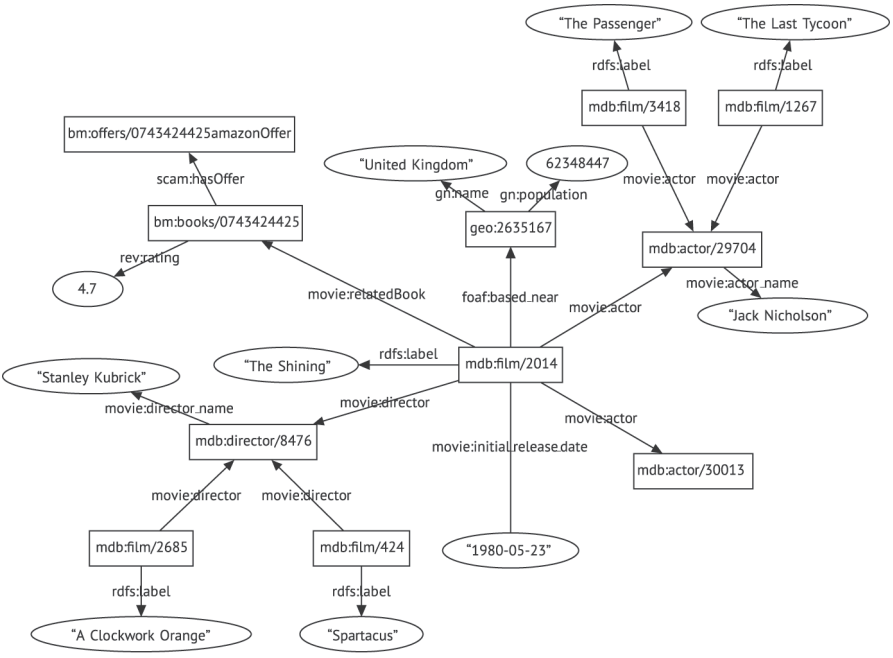


Рис. 12.16 ❖ RDF-граф, соответствующий набору данных на рис. 12.15

Множество троек называется *паттерном базового графа* (basic graph pattern – BGP), а выражения SPARQL, содержащие только их, называются *BGP-запросами*. Именно они являются предметом большинства работ по выполнению запросов в SPARQL.

Пример 12.15. В качестве примера рассмотрим SPARQL-запрос, который ищет имена фильмов, поставленных Стэнли Кубриком по книге с рейтингом больше 4.0:

```
SELECT ?name
WHERE {
?m rdfs:label ?name. ?m movie:director ?d.
?d movie:director_name "Stanley Kubrick".
?m movie:relatedBook ?b. ?b rev:rating ?r.
FILTER(?r > 4.0)
}
```

Здесь первые три строки во фразе **WHERE** образуют BGP, состоящий из пяти троек. С каждой тройкой связана *переменная*, например «?m», «?name» и «?r», а для переменной «?r» задан фильтр: **FILTER**(?r > 4.0). ♦

SPARQL-запрос можно также представить в виде *графа запроса*. Графом запроса называется семерка $Q = \langle V^Q, L_V^Q, E^Q, L_E^Q, f_V^Q, f_E^Q, FL \rangle$, где

1) $V^Q = V_c^Q \cup V_e^Q \cup V_l^Q \cup V_p^Q$ – множество вершин, соответствующих всем субъектам и объектам в SPARQL-запросе, где V_p^Q – множество вершин переменных, а V_c^Q, V_e^Q и V_l^Q – соответственно множества вершин классов, сущностей и литералов в графе запроса Q;
2) E^Q – множество ребер, соответствующих свойствам в SPARQL-запросе;
3) L_V^Q – множество меток вершин в Q, а L_E^Q – метки ребер в E^Q;
4) f_V^Q: $V^Q \rightarrow L_V^Q$ – биективная функция пометки вершин, которая сопоставляет каждой вершине в Q метку из L_V^Q. Меткой вершины $v \in V_p^Q$ является вершинная переменная, вершины $v \in V_l^Q$ – ее литеральное значение, а вершины $v \in V_c^Q \cup V_e^Q$ – соответствующий ей URI;
5) f_E^Q: $E^Q \rightarrow L_E^Q$ – биективная функция пометки ребер, которая сопоставляет каждому ребру в Q метку из L_E^Q. Меткой ребра может быть свойство или реберная переменная;
6) FL – фильтры ограничений.

Граф запроса Q_1 показан на рис. 12.17.

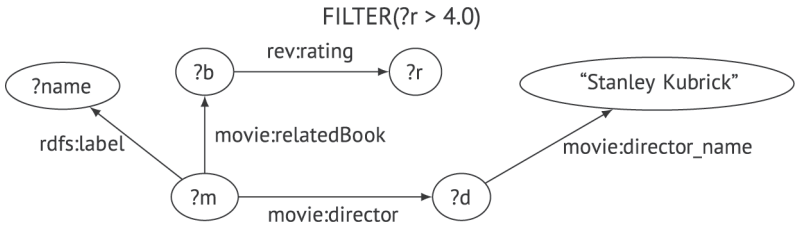


Рис. 12.17 ❖ Граф SPARQL-запроса, соответствующий запросу Q_1

Таким образом, семантика вычисления SPARQL-запроса определяется как отыскание всех подграфов RDF-графа G, гомоморфных графу запроса Q.

В разговорах о типах SPARQL-запросов часто упоминается форма графа запроса. Обычно встречаются три типа запросов: (i) линейные (рис. 12.18a), когда переменная на месте объекта в тройке встречается в субъекте другой тройки (например, ?y в *QL*); (ii) звездообразные (рис. 12.18b), когда переменная на месте объекта в тройке встречается в субъекте нескольких других троек (например, ?a в *QS*); (iii) в форме снежинки (рис. 12.18c), представляющей собой комбинацию нескольких звездообразных запросов.

Разработано несколько систем управления RDF-данными. Среди них можно выделить пять больших групп: отображающие RDF-данные непосредственно в реляционную систему; использующие реляционную схему с развитым индексированием (и «родной» системой хранения); денормализующие таблицу троек на кластерные свойства; использующие столбцовую организацию хранения; использующие собственную семантику сопоставления паттернов графов, присущую SPARQL.

Прямое реляционное отображение

В системах с прямым реляционным отображением используется тот факт, что тройки RDF имеют естественную табличную структуру. Поэтому создается одна таблица с тремя столбцами (Subject, Property, Object), в кото-

рой хранятся тройки (обычно есть и вспомогательные таблицы, но мы про них говорить не будем). Тогда SPARQL-запрос можно транслировать на SQL и применить к этой таблице. Показано, что язык SPARQL 1.0 допускает полную трансляцию на SQL. Вопрос о том, верно ли это для версии SPARQL 1.1 с ее дополнительными возможностями, пока остается открытым. Цель этого подхода – воспользоваться при выполнении SPARQL-запросов давно и хорошо разработанными методами хранения реляционных данных, обработки запросов и оптимизации. Он применяется в системах Sesame SQL92SAIL[1] и Oracle.

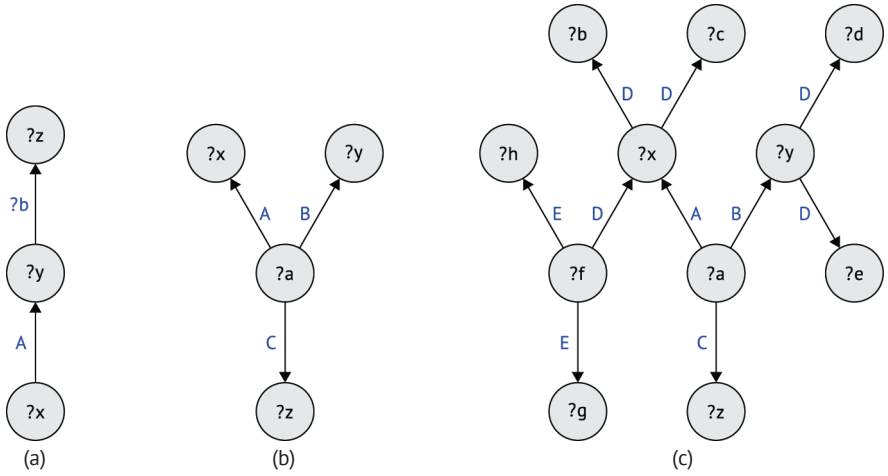


Рис. 12.18 ❖ Примеры форм SPARQL-запросов: (a) Q_L; (b) Q_S; (c) Q_K

Пример 12.16. В предположении, что таблица на рис. 12.15 реляционная, SPARQL-запрос из примера 12.15 можно транслировать на SQL следующим образом (здесь s, p, o соответствуют именам столбцов Subject, Property, Object):

```
SELECT T1.object
FROM T AS T1, T AS T2, T AS T3,
T AS T4, T AS T5
WHERE T1.p="rdfs:label"
AND T2.p="movie:relatedBook"
AND T3.p="movie:director"
AND T4.p="rev:rating"
AND T5.p="movie:director_name"
AND T1.s=T2.s
AND T1.s=T3.s
```

[1] Sesame способна взаимодействовать с любой системой хранения, поскольку реализует уровень хранения и логического вывода (Storage and Inference Layer – SAIL) для интерфейса с конкретной системой хранения, поверх которой она надстроена. SQL92SAIL – конкретная реализация для работы с реляционными системами.

```
AND T2.o=T4.s
AND T3.o=T5.s
AND T4.o > 4.0
AND T5.o="Stanley Kubrick"                                    ♦
```

Как видно из примера, этот подход приводит к большому числу соединений таблиц с собой, что нелегко поддается оптимизации. Кроме того, для больших наборов данных эта таблица троек становится очень большой, что дополнительно усложняет обработку запроса.

Одна таблица с развитым индексированием

Чтобы справиться с проблемами, свойственными прямому реляционному отображению, можно разработать платформенную систему хранения, допускающую развитое индексирование таблицы троек. Примерами такого подхода служат системы Hexastore и RDF-3X. Поддерживается только одна таблица, но над ней построено много индексов. Например, в RDF-3X создаются индексы для всех шести возможных перестановок субъекта, свойства и объекта: (spo, sop, ops, ops, sop, pos). Каждый индекс лексикографически отсортирован сначала по первому столбцу, потом по второму и, наконец, по третьему. Эти комбинации хранятся в листовых узлах кластерного B^+-дерева.

Преимущество такой организации в том, что SPARQL-запросы можно выполнить эффективно вне зависимости от того, где встречаются переменные (субъект, свойство, объект), поскольку один из индексов обязательно применим. Кроме того, она допускает индексную обработку запросов, которая устраняет необходимость некоторых самосоединений – они превращаются в запросы по диапазону к определенному индексу. Но даже когда соединения необходимы, можно воспользоваться быстрым соединением посредством слияния, поскольку каждый индекс отсортирован по первому столбцу. Недостатки тоже очевидны – потребление места на диске и накладные расходы на обновление нескольких индексов, если данные динамично изменяются.

Таблицы свойств

В подходе на основе таблиц свойств используется регулярность наборов данных RDF, в которых встречаются повторяющиеся паттерны. Поэтому «родственные» свойства хранятся в одной таблице. Впервые этот подход был применен в системе Jena, такая же стратегия используется в системе IBM DB2RDF. В обоих случаях результирующие таблицы отображаются на реляционную систему, а запросы транслируются на SQL.

В Jena определены два типа таблиц свойств. В таблицах первого типа – их можно назвать *кластерными таблицами свойств* – группируются свойства, которые часто встречаются в одинаковых (или похожих) субъектах. Определены разные табличные структуры для однозначных и многозначных свойств. Для однозначных свойств таблица содержит столбец субъекта и несколько столбцов свойств (рис. 12.19а). Значением свойства может быть null, если не существует тройки RDF, в которой встречается такой субъект и такое свойство. Каждая строка таблицы представляет несколько троек RDF – столько, сколько в ней имеется свойств, отличных от null. В этих таблицах субъект

является первичным ключом. Для многозначных свойств структура таблицы содержит субъект и многозначное свойство (рис. 12.19b). Каждая строка такой таблицы представляет одну тройку RDF; первичный ключ составной – (субъект, свойство). Отображение таблицы троек на таблицы свойств – проблема проектирования базы данных, которая решается администратором базы.

В Jena также определена *таблица классов свойств*, в которой кластеризованы субъекты с одним и тем же типом свойств (рис. 12.19c). В этом случае все члены класса (вспомните обсуждение структуры класса в контексте RDFS) собраны в одной таблице. Столбец «Type» содержит значение `rdf:type`, общее для всех свойств в этой строке.

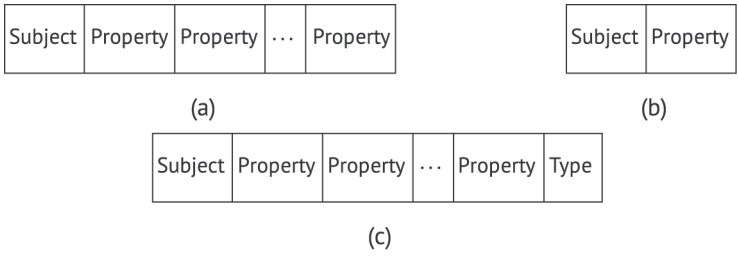


Рис. 12.19 ❖ Структура кластерной таблицы свойств

Пример 12.17. Набор данных из примера 12.14 можно организовать так, что одна таблица будет включать свойства субъектов, представляющих фильмы, другая – свойства режиссеров, третья – свойства актеров, четвертая – свойства книг и т. д. ♦

В системе IBM DB2RDF используется такая же стратегия, но организация таблиц более динамична (рис. 12.20). Таблица, называемая *прямым первичным хешем* (direct primary hash – DPH), организована по субъектам, но вместо того чтобы вручную идентифицировать «похожие» свойства, таблица включает k столбцов свойств, так что в одном и том же столбце двух разных строк могут храниться разные свойства. Каждый столбец свойства на самом деле состоит из двух столбцов: в одном находится метка свойства, в другом – его значение. Если число свойств для данного субъекта больше k, то дополнительные свойства размещаются во второй строке, которая помечается признаком в столбце «spill» (переполнение). Для многозначных свойств поддерживается таблица *прямого вторичного хеша* (direct secondary hash – DSH) – в качестве значения свойства в DPH хранится уникальный идентификатор *l_id*, а его настоящие значения находятся в строках таблицы DSH с таким идентификатором.

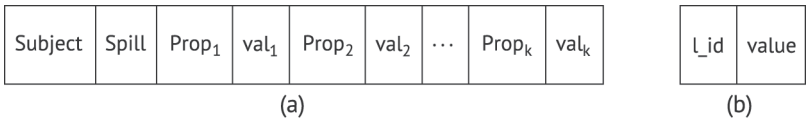


Рис. 12.20 ❖ Структура таблицы DB2RDF: (a) DPH; (b) DSH

Преимущество таблицы свойств заключается в том, что соединения в звездообразных запросах (т. е. соединения субъект-субъект) становятся просмотрами одной таблицы. Поэтому в оттранслированном запросе оказывается меньше соединений. Недостатки же в том, что при обоих подходах таблицы могут содержать много значений null, а многозначные свойства требуют специального обращения. Кроме того, хотя звездообразные запросы можно обработать эффективно, на запросы других типов это не распространяется. Наконец, если используется ручное назначение, то кластеризация «похожих» свойств оказывается нетривиальным занятием, а неправильные проектные решения могут усугубить проблему null-значений.

Двоичные таблицы

Подход на основе двоичных таблиц следует идее столбцовой организации базы данных – для каждого свойства определяется таблица с двумя столбцами: субъект и объект. В результате получается набор таблиц, упорядоченных по субъекту. Это типичная для столбцовых баз данных организация, преимущества которой, как и во всех подобных системах, – уменьшение объема ввода-вывода (поскольку читаются только нужные свойства) и длины кортежа, сжатие благодаря избыточности в значениях столбцов и т. д. К тому же в таблицах нет значений null, в отличие от таблиц свойств, а многозначные свойства поддерживаются естественно – каждому значению соответствует отдельная строка таблицы, как в случае таблицы DSH в DB2RDF. Кроме того, поскольку таблицы упорядочены по субъекту, соединения субъект-субъект можно эффективно реализовать с помощью слияния. Но есть и недостатки: в запросах больше операций соединения, в т. ч. типа субъект-объект, для которых слияние не поможет. Отметим также, что операция вставки сопровождается повышенными накладными расходами, потому что необходимо обновить несколько таблиц. Возражают, что проблему вставки можно сгладить, если делать это пакетно, но в динамичных репозиториях RDF сложность вставки все равно остается серьезной проблемой. Увеличение количества таблиц может негативно отразиться на масштабируемости (относительно числа свойств) этого подхода.

Пример 12.18. Представление набора данных из примера 12.14 в виде двоичных таблиц привело бы к созданию 18 таблиц – по одной для каждого свойства. Две из них показаны на рис. 12.21. ♦

Subject	Object
film/2014	“The Shining”
film/2685	“A Clockwork Orange”
film/424	“Spartacus”
film/1267	“The Last Tycoon”
film/3418	“The Passenger”
iso639-3/eng	“English”

(a)

Subject	Object
film/2014	actor/29704
film/2014	actor/30013
film/1267	actor/29704
film/3418	actor/29704

(b)

Рис. 12.21 ❖ Организация свойств демонстрационного набора данных: (a) rdfs:label и (b) movie:actor в виде двоичных таблиц (префиксы опущены)

Обработка на основе графов

В подходах к обработке RDF-данных на основе графов реализована исходная семантика RDF-запросов, определенная в начале этого раздела. Иными словами, поддерживается графовая структура RDF-данных (с использованием того или иного представления графа, например в виде списков смежности), SPARQL-запрос преобразуется в запрос к графу, для вычисления которого производится поиск гомоморфных подграфов в RDF-графе. Такой подход применяется в системах gStore и chameleon-db.

Преимущество этого подхода в том, что сохраняется исходное представление RDF-данных и гарантируется постулируемая семантика SPARQL. Недостаток – стоимость сравнения графов, поскольку задача о гомоморфизме графов NP-полная. В связи с этим встает вопрос о масштабируемости этого подхода на большие RDF-графы, для решения этой проблемы можно применить стандартные методы баз данных, в т. ч. индексирование. Далее мы опишем этот подход и проиллюстрируем возникающие проблемы на примере системы gStore.

В gStore для представления графов используются списки смежности. Вершины классов и сущностей кодируются битовой строкой фиксированной длины, в которой хранится информация о соседях. Эта информация используется в процессе сравнения графов. В результате генерируется *граф сигнатуры данных* G^*, каждая вершина которого соответствует вершине класса или сущности RDF-графа G. Точнее, G^* индуцирован всеми вершинами класса или сущности исходного RDF-графа G вместе с инцидентными им ребрами. На рис. 12.22a показан граф сигнатуры данных G^*, соответствующий RDF-графу G на рис. 12.16. Входной SPARQL-запрос также представлен *графом запроса* Q, который точно так же кодируется *графом сигнатуры запроса* Q^*. Граф сигнатуры запроса Q_2^*, соответствующий графу запроса на рис. 12.17, показан на рис. 12.22b.

Теперь задача сводится к нахождению соответствий Q^* в G^*. Хотя в результате кодирования RDF-граф и граф запроса уменьшились, NP-полнота задачи никуда не делась. Поэтому в gStore для уменьшения пространства поиска применяется стратегия фильтрации с вычислением. Цель состоит в том, чтобы сначала применить стратегию отсечения ложноположительных результатов и найти множество подграфов-кандидатов (обозначается CL), а затем проверить их, пользуясь списками смежности для нахождения множества окончательных ответов (обозначается RS). Таким образом, требуется решить две задачи. Во-первых, метод кодирования должен гарантировать, что $RS \subseteq CL$ – можно доказать, что описанная выше схема кодирования удовлетворяет этому условию. Во-вторых, нужен эффективный алгоритм сравнения подграфов, чтобы найти в G^* совпадения с Q^*. Для этого в gStore используется индексная структура, называемая VS*-деревом, которая дает граф, являющийся краткой сводкой G^*. VS*-дерево позволяет эффективно искать в G^* совпадения с Q^*, применяя стратегию отсечения ветвей для уменьшения пространства поиска.

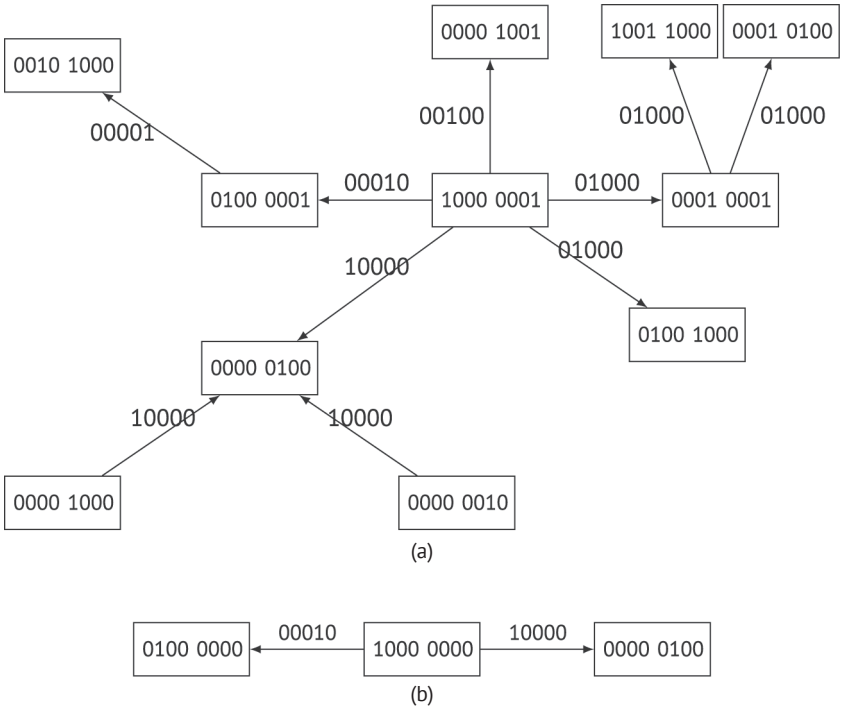


Рис. 12.22 ❖ Графы сигнатур:
(a) граф сигнатуры данных G^*; (b) граф сигнатуры запроса Q^*

Распределенное и федеративное выполнение SPARQL-запросов

По мере увеличения размеров RDF-наборов были разработаны методы горизонтального масштабирования, включающие параллельную и распределенную обработку. Во многих таких решениях RDF-граф G разбивается на несколько фрагментов, которые размещаются в разных узлах параллельной или распределенной системы. В каждом узле развернуто некоторое централизованное хранилище RDF. Во время выполнения SPARQL-запрос Q разлагается на несколько подзапросов, так чтобы на каждый подзапрос можно было получить ответ локально в одном узле, а затем результаты агрегируются. В разных работах предлагаются различные стратегии разбиения, которые, в свою очередь, приводят к различным методам обработки запросов. В некоторых подходах применяются решения на основе MapReduce, когда тройки RDF хранятся в HDFS и каждая тройка оценивается путем просмотра HDFS-файлов с последующей реализацией соединения в MapReduce. В других подходах применяются методологии распределенной или параллельной обработки запросов, подробно описанные в главах этой книги, – когда запрос разбивается на подзапросы и вычисляется в нескольких узлах.

Была также предложена альтернатива – использовать частичное вычисление запроса для выполнения распределенных SPARQL-запросов. Частичное вычисление функции – хорошо известная техника программирования со следующей идеей: пусть дана функция $f(s, d)$, где s – известный, а d – пока

еще неизвестный вход, тогда можно вычислить часть *f*, зависящую только от *s*, и получить частичный ответ. При таком подходе данные секционируются, а запросы нет – каждый узел получает полный SPARQL-запрос *Q* и выполняет его применительно к своему локальному фрагменту RDF-графа, так что вычисления распараллеливаются. В этом случае стратегия частичного вычисления применяется следующим образом: каждый узел S_i рассматривает фрагмент F_i как известный вход на своем этапе частичного вычисления; неизвестным входом является остальная часть графа ($\overline{G} = G \setminus F_i$). Необходимо разрешить две важные задачи. Первая – получить результаты частичного вычисления в каждом узле S_i для данного графа запроса *Q* – иными словами, решить задачу нахождения подграфов F_i, гомоморфных *Q*; она называется *локальным частичным сопоставлением*, потому что находит соответствия только во фрагменте F_i. Поскольку при вершинно-непересекающемся разбиении невозможно гарантировать, что и множества ребер не пересекаются, между фрагментами графа будут существовать *перекрестные ребра*. Вторая задача – собрать результаты локальных частичных сопоставлений и вычислить перекрестные сопоставления. Ее можно решить либо в управляющем узле, либо по аналогии с распределенным соединением.

В описанных выше подходах берется централизованный набор данных RDF и секционируется для распределенного или параллельного выполнения. Во многих задачах, относящихся к RDF, возникают проблемы, похожие на те, что мы обсуждали при интеграции баз данных, они требуют федеративного решения. В мире RDF некоторые узлы, содержащие RDF-данные, также способны обрабатывать SPARQL-запросы; они называются *оконечными точками SPARQL*. Типичным примером служит проект LOD, в котором различные RDF-репозитории взаимосвязаны и образуют *виртуально интегрированную распределенную базу данных*. В федеративных средах RDF часто предварительно вычисляют метаданные для каждой оконечной точки SPARQL. Метаданные могут содержать спецификацию возможностей оконечной точки или описание троек, т. е. свойств, о которых можно получить сведения в данной оконечной точке или еще какую-то информацию, необходимую конкретному алгоритму. На основе метаданных исходный SPARQL-запрос разлагается на несколько подзапросов, каждый из которых отправляется релевантным оконечным точкам SPARQL. Затем результаты подзапросов объединяются для получения ответа на исходный запрос.

Альтернативой предварительному вычислению метаданных является использование ASK-запросов на языке SPARQL для сбора информации о каждой оконечной точке и конструирования метаданных «на лету». По результатам этих запросов производится разложение SPARQL-запроса на подзапросы и назначение их оконечным точкам.

12.6.2.3. Навигация и опрос в проекте LOD

LOD состоит из множества *веб-документов*. Поэтому отправной точкой является веб-документ с включенными в него тройками RDF, которые кодируют веб-ресурсы. Тройки RDF содержат *ссылки на данные* в других документах, что позволяет связать веб-документы в граф.

Семантика SPARQL-запросов к LOD нетривиальна. Можно принять *семантику полного веба*, когда областью поиска при вычислении выражения SPARQL-запроса являются все связанные ссылками данные. Неизвестен завершающийся алгоритм выполнения запроса, который гарантировал бы полноту при такой семантике. Альтернатива – семейство *семантик на основе достижимости*, в которых область поиска определяется в терминах достижимых документов: если дано множество начальных URI и условие достижимости, то областью поиска являются все данные, до которых можно добраться из этих URI по путям, состоящим из ссылок на данные, и которые удовлетворяют условию достижимости. Конкретная семантика определяется условием достижимости. При такой постановке существуют практически пригодные алгоритмы.

Есть три подхода к выполнению SPARQL-запросов к LOD: на основе обхода, на основе индекса и гибридные. В системах *на основе обхода*, по существу, реализуется семантика на основе достижимости: стартовав из начальных URI, они рекурсивно находят релевантные URI, следуя по ссылкам на данные. Производительность таких алгоритмов сильно зависит от выбора начальных URI. Преимущество решений на основе обхода – в простоте реализации, поскольку не нужно поддерживать никакие структуры данных (например, индексы). Недостаток же – в длительном времени выполнения, поскольку эти алгоритмы просматривают веб-документы, а извлечение данных из каждого документа приводит к значительным задержкам. Кроме того, возможности их распараллеливания ограничены – они допускают распараллеливание ровно в той мере, в какой его допускают алгоритмы поискового робота.

В подходах *на основе индекса* для определения релевантных URI используется индекс, что уменьшает количество связанных документов, к которым необходимо обратиться. Разумным ключом индекса является тройка, тогда «релевантные» данному запросу URI можно определить из индекса, а запрос выполняется над данными, извлеченными в результате доступа к этим URI. В таких системах извлечение данных можно полностью распараллелить, что снижает негативное влияние извлечения данных на время выполнения запроса. Недостатки такого подхода – зависимость от индекса. Тут и задержка, связанная с построением индекса, и ограничения, налагаемые индексом на то, что именно может быть выбрано, и вопросы поддержания актуальности, проистекающие из динамичности веба.

При *гибридном подходе* производится обход с использованием приоритетного списка справочных URI. Начальные URI берутся из заранее построенного индекса, а вновь обнаруженные URI, отсутствующие в индексе, ранжируются согласно количеству ссылающихся на них документов.

12.6.3. Вопросы качества данных при интеграции веб-данных

В главе 7 (конкретно в разделе 7.1.5) мы обсуждали вопросы качества и очистки данных в системах интеграции баз данных (в основном в хранилищах данных). Вопрос о качестве в случае веб-данных стоит еще острее уже вследствие количества источников веб-данных, а также из-за неконтролируемого

процесса ввода данных в источнике и еще большего разнообразия данных. К качеству данных принято относить как непротиворечивость, так и достоверность данных (аутентичность и соответствие реальности). В хранилище данных непротиворечивость достигается с помощью процесса очистки данных, в ходе которого выявляются и устраняются ошибки и несогласованности. В контексте веба (а также озер данных) очистка осложняется отсутствием схемы и ограниченным количеством ограничений целостности, которые можно определить без схемы.

Проверка данных на достоверность остается серьезной проблемой. Однако если многие источники данных пересекаются, а так часто бывает с данными, поступающими из веба, то имеется значительная избыточность. Иногда можно применить эффективные методы слияния данных (мы обсудим их ниже), чтобы выявить правильные значения элементов данных, встречающихся в разных источниках, и тем самым установить истину.

В этом разделе мы осветим основные проблемы качества и очистки данных и обсудим существующие на данный момент решения.

12.6.3.1. Очистка структурированных веб-данных

Структурированные веб-данные составляют важную категорию данных в вебе, и им свойственны многочисленные проблемы с качеством. Сначала мы приведем краткий обзор методов очистки структурированных данных вообще, а затем отметим специфические проблемы, возникающие в вебе.

На рис. 12.23 показан типичный технологический процесс очистки структурированных данных, включающий три шага: факультативное обнаружение и профилирование, выявление ошибок и исправление ошибок. Для очистки «грязного» набора данных часто бывает необходимо смоделировать различные аспекты этих данных (собрать метаданные), например схему, паттерны, распределения вероятности и прочее. Для этого можно проконсультировать-

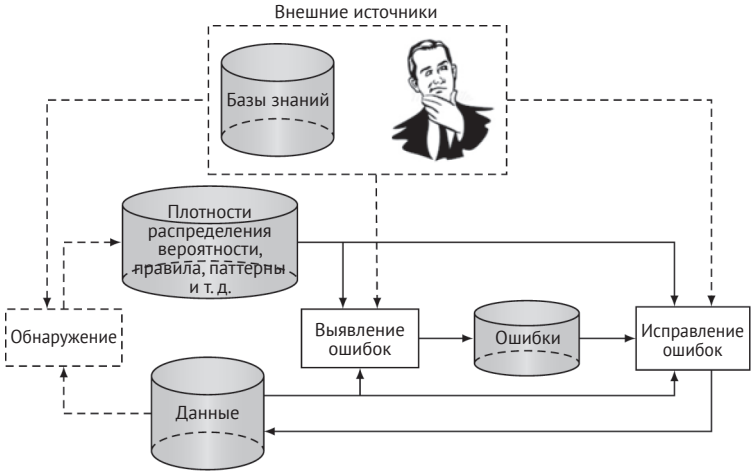


Рис. 12.23 ❖ Типичный технологический процесс очистки структурированных данных

ся со специалистами в предметной области, но обычно это дорогостоящий и занимающий много времени процесс, поэтому часто применяют автоматическую процедуру обнаружения и профилирования метаданных. При наличии грязного набора данных и ассоциированных с ним метаданных шаг выявления ошибок состоит в нахождении той части данных, которая не согласуется с метаданными, и объявлении этого подмножества ошибочным. Ошибки, найденные на этом шаге, могут принимать различные формы, например: выбросы, нарушения ограничений целостности, дубликаты. Наконец, на шаге исправления данных к грязному набору применяются обновления, которые устраняют замеченные ошибки. Поскольку в процессе очистки данных много неопределенностей, для повышения его точности при всяком возможном случае производятся обращения к внешним источникам, например базам знаний и специалистам.

Описанный процесс хорошо работает для структурированных таблиц с богатым набором метаданных, например имеющим большую схему с достаточным количеством ограничений, чтобы промоделировать взаимодействия строк и столбцов. Кроме того, процесс очистки и выявления ошибок работает лучше, если имеется достаточное количество примеров (кортежей), чтобы алгоритмы могли автоматически сравнить разные экземпляры и определить, где могут быть ошибки. Но для веб-таблиц ни одно из этих условий не выполняется, поскольку большинство таблиц короткие (всего несколько кортежей) и узкие (атрибутов немного). Хуже того, количество веб-таблиц намного больше, чем в хранилище данных. Это означает, что ручная очистка, теоретически мыслимая для одной веб-таблицы, для всех структурированных веб-таблиц практически неосуществима. На рис. 12.24 приведено несколько примеров ошибок в таблицах из Википедии, а согласно некоторым оценкам всего таких ошибок примерно 300 тысяч.

Sevilla - Jerez de la Frontera-C´adiz	1861
C`ordoba - M´alaga	1865.
Bobadilla - Granada	1874
C`ordoba - B´elmez	1874
Osuna	La Roda

(a)

Polaco	15.04.1983	194	84
Vini	29.09.1982	N/A	N/A
Caiao	30/11/1982	N/A	N/A
Jairo	17.02.1990	N/A	N/A
Michael	20.04.1983	N/A	N/A
Ricardinho	19.11.1975	192	94

(b)

2002[12]	10.300 oz	899,500 oz
2005[13]	25.272	2.174.620 oz
2006[13]	49.354 oz	3.005.611 oz
2007[13]	48.807 oz	3.165408 oz
2008[9]	47.755 oz	3.157.837 oz
2009[2]	0.9 million oz	818.050 oz

(c)

WARRIORS@Susses Thunder	13-28	–
WARRIORS@Hampshire Thrashers	42-13	–
Essex Spartans@WARRIORS	P-P	Postponed
WARRIORS@Cambridgeshire Cats	36-44	–
East Kent Mavericks@WARRIORS	12-18	–
WARRIORS@East Kent Mavericks	15-17	–

(d)

Рис. 12.24 ❖ Проблемы качества структурированных веб-данных (ошибочные данные показаны красным цветом): (a) лишняя точка; (b) даты в разных форматах; (c) несогласованные единицы измерения веса; (d) отсутствует счет матча. По материалам работы [Huang and He 2018]

12.6.3.2. Слияние веб-данных

При интеграции веб-данных часто возникает задача слияния данных (data fusion), т. е. принятие решения о том, какое из значений некоторого элемента данных, имеющего разные представления в нескольких источниках, правильно. Проблема в том, что представления в разных источниках веб-данных могут быть противоречивы, что затрудняет слияние. Существует два типа конфликтов: *неопределенность* и *противоречие*. Неопределенность – это конфликт между отличным от null значением и одним или несколькими null-значениями, описывающими одно и то же свойство реальной сущности. Причиной неопределенности является отсутствие информации, которое обычно представляется значением null. Противоречие – это конфликт между двумя или более разными отличными от null значениями одной и той же реальной сущности. Причина его в том, что разные источники дают разные значения одного и того же атрибута.

Таким образом, автоматическая очистка веб-таблицы – исключительно трудное дело. Хотя способы очистки, созданные для хранилищ данных, можно применить для исправления некоторых ошибок, в общем случае для очистки веб-таблиц нужны более изощренные методы. Из недавних систем, нацеленных на выявление ошибок в веб-таблицах, отметим Auto-Detect. Это управляемый данными статистический метод, в котором используется статистика совместной встречаемости, набранная на большом корпусе данных. В его основе лежит предположение, что если некоторая комбинация значений встречается очень редко (для численной оценки используется поточечная взаимная информация), то можно заподозрить потенциальную ошибку. Хотя Auto-Detect способен выявить много ошибок, он не предлагает никаких исправлений. Решения, которые автоматически исправляют ошибки в веб-таблицах (или хотя бы предлагают исправления), – дело будущего.

На рис. 12.25 показана классификация различных стратегий слияния. Стратегии *игнорирования конфликтов* просто передают конфликт на рассмотрение пользователю или приложению. Стратегии *предотвращения конфликтов* признают существование конфликтующих представлений и применяют простое правило для принятия однозначного решения на основе экземпляров данных или метаданных. Пример такой стратегии на основе экземпляров –

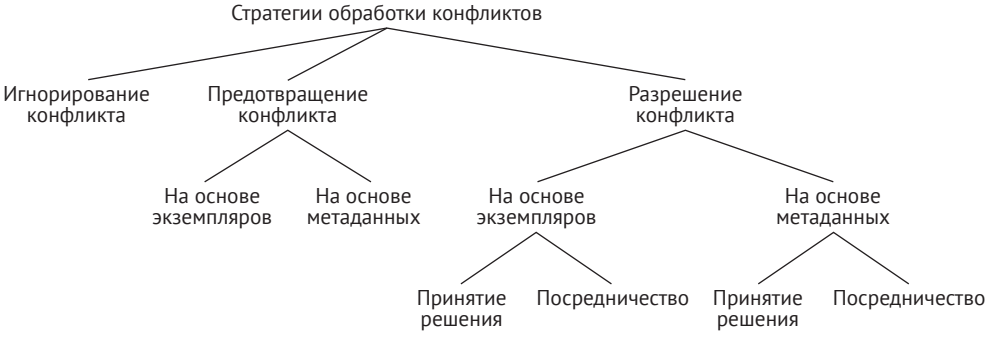


Рис. 12.25 ❖ Классификация стратегий слияния данных. По материалам работы [Bleiholder and Naumann 2009]

предпочесть отличное от null значение значениям null. Пример стратегии на основе метаданных – предпочесть значение из одного источника значению из другого. Стратегии *разрешения конфликтов* выбирают значение из уже присутствующих (принятие решение) или значение, которого может не быть среди присутствующих (посредничество). Пример стратегии разрешения конфликтов путем принятия решения на основе экземпляров – брать наиболее частое значение. Пример посреднической стратегии на основе экземпляров – брать среднее всех присутствующих значений.

12.6.3.3. Качество источника веб-данных

Описанные выше стратегии разрешения конфликтов полагаются главным образом на сами значения и могут оказаться непригодными в трех отношениях. Во-первых, источники веб-данных бывают разного качества; данные, поступающие из более надежного источника, обычно точнее. Однако и надежные источники могут поставлять неправильные значения, поэтому часто требуется продвинутая стратегия разрешения, которая учитывает качество источника при предсказании правильного значения. Во-вторых, источники веб-данных могут копировать данные друг у друга, и игнорирование такого рода зависимостей может привести к неверным решениям. Например, стратегия разрешения конфликтов, основанная на мажоритарном голосовании, будет ошибаться, если часть данных скопирована. В-третьих, правильное значение элемента данных может изменяться со временем (например, место работы человека), поэтому, оценивая точность источника и принимая решения, очень важно различать неправильное и *устаревшее* значения.

Один из элементов продвинутой стратегии слияния данных – оценка надежности или качества источника. В этом разделе мы обсудим, как моделируется точность источника данных и как эта модель обобщается, чтобы учесть зависимости между источниками и актуальность данных.

Точность источника

Точность источника S, обозначаемая $A(S)$, измеряется как доля правильных значений, получаемых от него. Можно считать, что точность равна вероятности того, что значение, поставляемое S, правильно. Обозначим $V(S)$ множество значений, поставляемых S. Для каждого $v \in V(S)$ обозначим $Pr(v)$ вероятность того, что v – правильное значение. Тогда $A(S)$ вычисляется по формуле

$$A(S) = Avg_{v \in V(S)} Pr(v).$$

Рассмотрим элемент данных D. Обозначим $Dom(D)$ множество значений D, среди которых одно правильное и n неправильных. Пусть S_D – множество источников, поставляющих значение D, а $S_D(v) \subseteq S_D$ – множество источников, поставляющих конкретное значение v. Обозначим $\Phi(D)$ – множество наблюдаемых значений D, поставляемых всеми источниками $S \in S_D$. Вероятность $Pr(v)$ можно вычислить следующим образом:

$$Pr(v) = Pr(v \text{ является правильным значением} \mid \Phi(D)) \propto Pr(\Phi(D) \mid v \text{ является правильным значением}).$$

В предположении, что источники независимы и что n неправильных значений равновероятны, $Pr(\Phi(D) \mid v$ является правильным значением) можно вычислить следующим образом:

$$Pr(\Phi(D) \mid v \text{ является правильным значением}) = \prod_{S \in S_D(v)} A(S) \prod_{S \in S_D \setminus S_D(v)} \frac{1 - A(S)}{n},$$

или

$$Pr(\Phi(D) \mid v \text{ является правильным значением}) = \prod_{S \in S_D(v)} \frac{nA(S)}{1 - A(S)} \prod_{S \in S_D} \frac{1 - A(S)}{n}.$$

Поскольку величина $\prod_{S \in S_D} \frac{1 - A(S)}{n}$ одинакова для всех значений v, имеем:

$$Pr(\Phi(D) \mid v \text{ является правильным значением}) \propto \prod_{S \in S_D(v)} \frac{nA(S)}{1 - A(S)}.$$

Определим *число голосов* источника S следующим образом:

$$C(S) = \ln \frac{nA(S)}{1 - A(S)},$$

а *число голосов* значения v:

$$C(v) = \sum_{S \subseteq S_D(v)} C(S).$$

Интуитивно понятно, что чем больше число голосов источника, тем он точнее, а у значения с большим числом голосов больше шансов оказаться правильным. Подводя итог проведенному выше анализу, мы можем выписать для вычисления вероятности значения v:

$$Pr(v) = \frac{\exp(C(v))}{\sum_{v_0 \in Dom(v)} \exp(C(v_0))}.$$

Очевидно, что для элемента данных в качестве правильного должно быть выбрано значение $v \in Dom(D)$ с наибольшей вероятностью $Pr(v)$. Как видим, вычисление точности источника $A(S)$ зависит от вероятности $Pr(v)$, а вычисление вероятности $Pr(v)$ зависит от точности источника $A(S)$. Можно спроектировать алгоритм, который в начале работы назначает всем источникам одинаковую точность, а всем значениям – одинаковую вероятность, а затем итеративно вычисляет точности и вероятности, пока не будет достигнута сходимость. Считается, что алгоритм сошелся, если точности источников и найденные правильные значения перестали изменяться.

Зависимость источников друг от друга

В приведенном выше вычислении точности источников предполагается, что все источники независимы. В действительности источники копируют друг у друга данные, что создает зависимости. Для выявления факта копирования

можно применить два интуитивных соображения. Во-первых, у конкретного элемента данных существует только одно правильное значение, но обычно бывает несколько неправильных. Если два источника дают одно и то же правильное значение, то это еще не значит, что они зависимы, но если они дают одинаковые неправильные значения, то есть все основания заподозрить зависимость. Во-вторых, в случайном подмножестве значений, возвращаемых источником данных, точности обычно будут примерно такими же, как в полном множестве значений от этого источника. Но если источник занимается копированием, то точности скопированных данных могут отличаться от точностей данных, которые он возвращает независимо. Таким образом, если имеются два источника, один из которых копирует данные из другого, то источник, для которого точности собственных данных заметно отличаются от точностей данных, общих с другим источником, вероятно, и повинен в копировании. Исходя из этих соображений, можно разработать байесовскую модель для вычисления вероятности копирования между источниками S_1 и S_2 при условии наблюдений Ф над всеми элементами данных; затем найденные вероятности используются, чтобы скорректировать вычисление числа голосов за значение $C(v)$ с учетом зависимостей источников.

Актуальность источника

До сих пор мы предполагали, что слияние производится на статическом моментальном снимке данных. Но в действительности данные изменяются со временем, и правильное значение тоже может меняться. Например, время вылета рейса может меняться в зависимости от месяца, человек может переходить с одного места работы на другое, у компании может смениться генеральный директор. Для отслеживания таких изменений данные в источниках должны обновляться. В такой динамической постановке ошибки могут быть вызваны следующими причинами: (1) источники могут давать неправильные значения, как в статическом случае; (2) данные в источниках могут вообще не обновляться; (3) некоторые источники не обновляют данные своевременно. В этом контексте цель слияния данных – найти все правильные значения и периоды их правильности, если данные изменяются со временем. Если в статическом случае качество источника можно оценить с помощью точности, то в динамической постановке метрики для оценки качества сложнее – высококачественный источник должен давать новое значение элемента данных *тогда и только тогда, когда* это значение стало правильным, и притом *сразу после* этого события. Для оформления этих интуитивных соображений применяются три метрики: *покрытие* источника измеряет улавливаемое изменение значений различных элементов данных, *аккуратность* измеряет долю изменений, которые источник не улавливает (т. е. возвращает неправильное значение), *актуальность* измеряет скорость улавливания изменений источником. И снова можно применить байесовский анализ, чтобы оценить время каждого изменения элемента данных и его новое значение.

Для слияния данных и моделирования качества источников применялись также машинное обучение и вероятностные модели. В частности, в системе SLiMFast задача о слиянии данных ставится как задача статистического

обучения дискриминантных вероятностных моделей. В отличие от предыдущих подходов к слиянию, основанных на обучении, SLiMFast дает гарантии качества результатов слияния и может включать в процесс имеющиеся знания о предметной области. На рис. 12.26 приведена схема работы SLiMFast. На вход SLiMFast подаются (1) набор наблюдений над источниками, а точнее потенциально конфликтующие значения, возвращенные разными источниками для различных объектов; (2) факультативный размеченный набор правильных значений для подмножества объектов; (3) предметные знания об источниках, которые пользователи считают полезными для оценки их точности. На основе этой информации SLiMFast строит вероятностную графическую модель всестороннего обучения и логического вывода. В зависимости от объема доступных объективно правильных значений SLiMFast решает, какой алгоритм (EM или минимизации эмпирической потери) использовать для обучения параметров графической модели. Затем обученная модель используется для вывода значений объектов и точности источников.

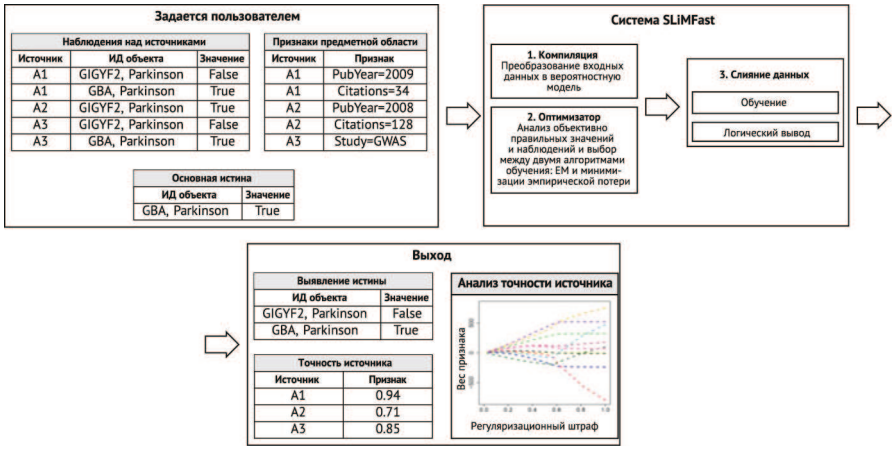


Рис. 12.26 ❖ Схема работы SLiMFast.
По материалам работы [Rekatsinas et al. 2017]

12.7. Библиографические замечания

На темы, связанные с вебом, существует много хороших источников информации, различающихся взглядом на проблему. В работе [Abiteboul et al. 2011] упор сделан на использовании XML и RDF для моделирования веб-данных, там же имеется обсуждение поиска и технологий больших данных, в частности MapReduce. Описание хранилищ веб-данных приведено в работе [Bhowmick et al. 2004]. Работа [Bonato 2008] акцентирована на моделировании веба в виде графа и исследованиях этого графа. Обзор ранних работ по языкам веб-запросов имеется в статье [Abiteboul et al. 1999].

Хороший обзор по проблемам веб-поиска см. в работе [Arasu et al. 2001], мы следовали ей в разделе 12.2. Работа [Lawrence and Giles 1998] содержит одно из первых обсуждений этой темы с упором на открытый веб. В работе [Florescu et al. 1998] проблематика веб-поиска рассматривается с точки зрения баз данных. Глубинный (скрытый) веб – тема работы [Raghavan and Garcia-Molina 2001]. В работах [Lage et al. 2002] и [Hedley et al. 2004b] также обсуждается поиск в глубинном вебе и приводится анализ результатов. Метапоиску для доступа к глубинному вебу посвящены работы [Ipeirotis and Gravano 2002, Callan and Connell 2001, Callan et al. 1999, Hedley et al. 2004a]. Проблема выбора баз данных в связи с метапоиском обсуждается в работах [Ipeirotis and Gravano 2002] и [Gravano et al. 1999] (GlOSS algorithm).

Статистические сведения об открытом вебе взяты из работы [Bharat and Broder 1998, Lawrence and Giles 1998, 1999, Gulli and Signorini 2005], а о глубинном вебе – из работ [Hirate et al. 2006] и [Bergman 2001].

Графовая структура веба и использование графов для моделирования и опроса веба – предмет многочисленных публикаций: в работах [Kumar et al. 2000, Raghavan and Garcia-Molina 2003, Kleinberg et al. 1999] обсуждается моделирование веб-графа, в работах [Kleinberg et al. 1999, Brin and Page 1998, Kleinberg 1999] – применение графов для поиска, а в работе [Chakrabarti et al. 1998] – для категоризации и классификации содержимого веба. Обсуждению характеристик веб-графа и сходства его структуры с галстуком-бабочкой посвящены работы [Bonato 2008], [Broder et al. 2000] и [Kumar et al. 2000]. Мы не стали обсуждать важные вопросы, относящиеся к управлению очень большим, динамично изменяющимся веб-графом. Это выходит за рамки главы, но все же упомянем два направления исследований. Первое – сжатие веб-графа для более эффективного хранения и манипуляций [Adler and Mitzenmacher 2001], второе – специальное представление веб-графа с помощью так называемых s-узлов [Raghavan and Garcia-Molina 2003].

Вопросам обхода веба роботом посвящены работы [Cho et al. 1998, Najork and Wiener 2001] и [Page et al. 1998], причем последняя является классической статьей по PageRank, а та модифицированная форма, которую мы обсуждали в этой главе, взята из работы [Langville and Meyer 2006]. Альтернативные подходы к задаче обхода веба рассматриваются в работах [Cho and Garcia-Molina 2000] (на основе изменения частоты), [Cho and Ntoulas 2002] (на основе выборки) и [Edwards et al. 2001] (инкрементный). Применение методов классификации для оценки релевантности обсуждается в работах [Mitchell 1997, Chakrabarti et al. 2002] (наивный байесовский классификатор), [Passerini et al. 2001], [Altingövde and Ulusoy 2004] (обобщения байесовского классификатора) и [McCallum et al. 1999], [Kaelbling et al. 1996] (обучение с подкреплением).

Индексирование веба – важная тема, которой мы посвятили раздел 12.2.2. Различные методы индексирования текста обсуждаются в работах [Manber and Myers 1990] (суффиксные массивы), [Hersh 2001, Lim et al. 2003] (инвертированные индексы) и [Faloutsos and Christodoulakis 1984] (файлы сигнатур). Книга [Salton 1989] – классический источник информации по обработке и анализу текстов. Проблемы построения инвертированных индексов для веба и их возможные решения обсуждаются в работах [Arasu et al. 2001], [Melnik et al. 2001] и [Ribeiro-Neto and Barbosa 1998]. Связанная с этим тема

ранжирования также стала предметом активных исследований. Помимо хорошо известного алгоритма PageRank, в работе [Kleinberg 1999] предложен алгоритм HITS.

При обсуждении слабо структурированного подхода к опросу веба мы иллюстрировали идеи и понятия на примере модели OEM и языка Lorel. Они осуждаются в работах [Papakonstantinou et al. 1995] и [Abiteboul et al. 1997]. Применение гидов по данным для упрощения OEM – тема работы [Goldman and Widom 1997]. Язык UnQL [Buneman et al. 1996] концептуально похож на Lorel. При обсуждении языков веб-запросов в разделе 12.3.2 мы выделили языки первого и второго поколений; это деление основано на работе [Florescu et al. 1998]. К языкам первого поколения относятся WebSQL [Mendelzon et al. 1997], W3QL [Konopnicki and Shmueli 1995] и WebLog [Lakshmanan et al. 1996], к языкам второго поколения – WebOQL [Arocena and Mendelzon 1998] и StruQL [Fernandez et al. 1997]. Говоря о вопросно-ответном подходе, мы упомянули следующие системы: Mulder [Kwok et al. 2001], WebQA [Lam and Özsu 2002], Start [Katz and Lin 2002] и Tritus [Agichtein et al. 2004].

Компоненты семантического веба представлены в работе [Antoniou and Plexousakis 2018]. Видение, воплощенное в проекте Linked Open Data (LOD) и его требованиях, обсуждается в работах [Bizer et al. 2018] и [Berners-Lee 2006]. Описанное в LOD разделение по тематике намечено в работе [Schmachtenberg et al. 2014].

Наше обсуждение RDF основано на работе [Özsu 2016]. Описано пять подходов к управлению RDF-данными: (1) прямое реляционное отображение – в работах [Angles and Gutierrez 2008], [Sequeda et al. 2014] обсуждается отображение SPARQL на SQL, а в работах [Broekstra et al. 2002] и [Chong et al. 2005] – системы Sesame SQL92SAIL и Oracle соответственно; (2) использование одной таблицы с развитым индексированием – Hexastore [Weiss et al. 2008] и RDF-3X [Neumann and Weikum 2008, 2009]); (3) таблицы свойств – Jena [Wilkinson 2006], IBM DB2RDF [Bornea et al. 2013]; (4) двоичные таблицы – система SW-Store [Abadi et al. 2009], основанная на предложении из работы [Abadi et al. 2007], а проблемы, связанные с большим количеством таблиц, обсуждаются в работе [Sidirourgos et al. 2008]; (5) на основе графов – [Bönström et al. 2003], gStore [Zou et al. 2011, 2014] и chameleon-db [Aluç 2015]). Методы на основе графов подробно обсуждаются в работе [Zou and Özsu 2017]. Распределенное и облачное выполнение SPARQL рассмотрено в работе [Kaoudi and Manolescu 2015]. Для выполнения SPARQL-запросов в проекте LOD [Hartig 2013a] есть три подхода: на основе обхода [Hartig 2013b, Ladwig and Tran 2011], на основе индекса [Umbrich et al. 2011] и гибридный [Ladwig and Tran 2010].

Очистка структурированных данных активно изучалась в контексте интеграции хранилищ данных [Rahm and Do 2000] и [Ilyas and Chu 2015]. Обобщение на более широкий контекст, в т. ч. веб, рассматривается в работе [Ilyas and Chu 2019]. Наше изложение слияния данных (раздел 12.6.3.2) и выделения двух типов конфликтов данных – неопределенность и противоречие – следует работе [Dong and Naumann 2009]. Обсуждение системы Auto-Detect в том же разделе заимствовано из работы [Huang and He 2018], а обсуждение классификации (как и рис. 12.25) – из работы [Bleiholder and Naumann 2009]. Моделирование точности источников данных, а также обобщение на вза-

имозависимые источники и вопросы актуальности источников изложены по материалам работ [Dong et al. 2009b, a]. За более полным рассмотрением темы слияния данных отсылаем читателя к пособию [Dong and Naumann 2009] и книге [Dong and Srivastava 2015]. Система SlimFAST (см. рис. 12.26) описана в работах [Rekatsinas et al. 2017] и [Koller and Friedman 2009].

Одна из первых систем очистки данных в озерах данных, CLAMS [Farid et al. 2016], позволяет выявлять ограничения целостности в данных, хранящихся в очереди, и контролировать их соблюдение. В CLAMS используется графовая модель данных, основанная на RDF, и новый формализм ограничений целостности, который позволяет задавать как реляционные ограничения, так и более выразительные правила контроля качества на основе графовых паттернов в виде ограничений отрицания [Chu et al. 2013]. В CLAMS также используется Spark и параллельные алгоритмы для проверки ограничения и обнаружения несогласованных данных.

Упражнения

Задача 12.1. Чем поиск в вебе отличается от запросов к вебу?

Задача 12.2 (**). Рассмотрим общую архитектуру поисковой системы на рис. 12.2. Предложите архитектуру веб-сайта с кластером без разделения ресурсов, которая реализует все компоненты, показанные на этом рисунке, и плюс веб-серверы в среде, где имеется очень много веб-документов, очень большие индексы и очень много веб-пользователей. Определите, как следует секционировать и реплицировать множество веб-страниц в каталоге страниц и индексы. Обсудите основные преимущества своей архитектуры с точки зрения масштабируемости, отказоустойчивости и производительности.

Задача 12.3 (**). Продолжим задачу 12.2. Рассмотрим запрос от веб-клиента поисковой системы на поиск по ключевым словам. Предложите параллельную стратегию выполнения запроса, которая ранжирует результирующие страницы и показывает реферат каждой страницы.

Задача 12.4 (*). Чтобы повысить локальность доступа и производительность системы в различных географических регионах, предложите расширение архитектуры сайта из задачи 12.3 на несколько сайтов, реплицирующих все страницы. Определите, как осуществляется репликация веб-страниц. Определите, как запрос пользователя маршрутизируется к конкретному сайту. Обсудите преимущества своей архитектуры с точки зрения масштабируемости, доступности и производительности.

Задача 12.5 (*). Продолжим задачу 12.4. Рассмотрим запрос от веб-клиента поисковой системы на поиск по ключевым словам. Предложите параллельную стратегию выполнения запроса, которая ранжирует результирующие страницы и показывает реферат каждой страницы.

Задача 12.6 (**). Рассмотрим два источника веб-данных, которые моделируются как отношения `EMP1`(Name, City, Phone) и `EMP2`(Firstname, Lastname,

City). Предположим, что после интеграции схем определено представление EMP(Firstname, Name, City, Phone) над EMP1 и EMP2, где каждый атрибут EMP совпадает либо с атрибутом EMP1, либо с атрибутом EMP2, а EMP2.Lastname переименован в Name. Обсудите ограничения такой интеграции. Теперь предположим, что оба источника поставляют данные в формате XML. Приведите определения XML-схем EMP1 и EMP2. Предложите XML-схему, которая интегрирует EMP1 и EMP2 и не страдает от проблем, свойственных отношению EMP.

Предметный указатель

И

К

Л

M

H

O

П

Р

С

Т

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

М. Тамер Ёсу, Патрик Вальдуриес

Принципы организации распределенных баз данных

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура PT Serif. Печать офсетная.
Усл. печ. л. 54,6. Тираж 200 экз.

Отпечатано в ПАО «Т8 Издательские Технологии»
109316, Москва, Волгоградский проспект, д. 42, корпус 5

Веб-сайт издательства: **www.dmkpress.com**