O'REILLY®

Конвейеры данных. Карманный справочник

Конвейеры данных — это фундамент успеха в анализе данных. Сбор данных из множества разнообразных источников и преобразование их для использования в контексте задачи — вот где кроется разница между наличием данных и получением от них реальной пользы. Этот карманный справочник дает определение конвейеров данных и объясняет, как они работают в современном стеке данных.

Вы познакомитесь с общими соображениями и ключевыми моментами принятия решений при реализации конвейеров, таких как пакетный или потоковый прием данных, а также выбор между разработкой и покупкой инструментов. В книге рассматриваются наиболее распространенные решения, принимаемые специалистами по данным, и обсуждаются основополагающие концепции, применимые к платформам с открытым исходным кодом, коммерческим продуктам и к собственным разработкам.

- **Что такое конвейер данных и как он работает**
- **Как данные перемещаются и обрабатываются в современной инфраструктуре, включая облачные платформы**
- **Популярные инструменты и продукты, применяемые для построения конвейеров**
- **Как конвейеры помогают закрыть потребности в аналитике и отчетности**
- **Вопросы обслуживания, тестирования и предупреждения сбоев конвейеров данных**

ISBN 978-601-09-2561-8

O'REILLY®

Конвейеры данных. Карманный справочник

Сбор и обработка данных для аналитики

Джеймс Денсмор

Data Pipelines Pocket Reference

Moving and Processing Data for Analytics

James Densmore

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Джеймс Денсмор

Конвейеры данных. Карманный справочник

Астана
«АЛИСТ»
2024

УДК 004.42
ББК 32.973.26-018.1
Д33

Денсмор Дж.

Д33 Конвейеры данных. Карманный справочник: Пер. с англ. — Астана: АЛИСТ, 2024. — 256 с.: ил.

ISBN 978-601-09-2561-8

Книга посвящена передовым методам построения конвейеров данных, сбору данных из множества разнообразных источников и преобразованию их для аналитики. Дано введение в конвейеры данных, раскрыта их работа в современном стеке данных. Описаны стандартные шаблоны конвейеров данных. Показан процесс сбора данных от их извлечения до загрузки в хранилище. Затронуты вопросы преобразования и проверки данных, оркестровки конвейеров, методов их обслуживания и мониторинга производительности. Примеры программ написаны на Python и SQL и задействуют множество библиотек с открытым исходным кодом.

Для специалистов по обработке данных

УДК 004.42
ББК 32.973.26-018.1

ISBN 978-1-492-08783-0 (англ.)
ISBN 978-601-09-2561-8 (каз.)

Оглавление

Предисловие

Конвейеры данных — это основа успеха в области анализа данных и машинного обучения. Сбор данных из многочисленных и разнообразных источников и их обработка для обеспечения контекста — вот в чем заключается разница между простым наличием данных и получением пользы от них.

Я работаю аналитиком данных, инженером данных[1] и руководителем команд в области анализа данных более 10 лет. За прошедшее время я стал свидетелем быстрых изменений и роста в этой области. Появление облачной инфраструктуры и, в частности, облачных хранилищ данных позволяет переосмыслить способ проектирования и реализации конвейеров данных.

Эта книга описывает передовые методы построения конвейеров данных в современную эпоху. Я основываю свои советы и наблюдения на собственном опыте, а также на опыте лидеров отрасли, которых я знаю и за которыми следую.

Моя цель состоит в том, чтобы эта книга стала для вас путеводителем и карманным справочником. Хотя ваши потребности во многом зависят от вашей организации и задач, которые вы намереваетесь решить, я много раз добивался успеха, ограничиваясь лишь основами, изложенными в этой книге. Я надеюсь, что вы воспримете эту книгу как ценный ресурс, помогающий строить и поддерживать конвейеры данных в вашей организации.

Для кого эта книга

Основная аудитория этой книги — опытные и начинающие специалисты по обработке данных, а также члены аналитических

---

[1] Здесь и далее принят дословный перевод англ. термина data engineer. — *Ред.*

групп, которые хотят понять, что такое конвейеры данных и как они реализуются. В их число входят инженеры данных, технические руководители, специалисты по хранилищам данных, инженеры-аналитики, специалисты по бизнес-аналитике и руководители аналитиков на уровне директора/вице-президента.

Я предполагаю, что у вас есть базовые представления о концепциях хранилищ данных. Для реализации обсуждаемых примеров вы должны свободно работать с базами данных SQL, REST API и форматом JSON. Вы должны владеть как минимум одним скриптовым языком, например Python. В идеале желательно обладать базовыми навыками работы с командной строкой Linux и хотя бы одной платформой облачных вычислений.

Все примеры программного кода написаны на Python и SQL и задействуют множество библиотек с открытым исходным кодом. Я использую Amazon Web Services (AWS) для демонстрации методов, описанных в книге, и службы AWS применяются во многих примерах кода. Когда это возможно, я обращаю внимание на аналогичные услуги других крупных облачных провайдеров, таких как Microsoft Azure и Google Cloud Platform (GCP). Все примеры кода можно изменить для выбранного вами облачного провайдера, а также для локального использования.

Условные обозначения, используемые в этой книге

В этой книге приняты следующие типографские соглашения:

Курсив — указывает новые термины.

Полужирный шрифт — выделяет интернет-ссылки и адреса электронной почты.

`Моноширинный шрифт` — используется для листингов программ, а также внутри абзацев для ссылки на элементы программы, такие как имена переменных или функций, типы данных, переменные среды, операторы и ключевые слова.

`Полужирный моноширинный шрифт` — выделяет команды или другой текст, который пользователь должен ввести самостоятельно.

Курсивный моноширинный шрифт — показывает текст, который следует заменить значениями, заданными пользователем, или значениями, определенными контекстом.

Скачивание примеров кода

Дополнительные материалы (примеры кода, упражнения и т. д.) доступны для загрузки по адресу:

https://oreil.ly/datapipelinescode.

Если у вас есть технический вопрос или проблема с использованием примеров кода, отправьте электронное письмо по адресу:

bookquestions@oreilly.com.

Эта книга предназначена для того, чтобы помочь вам выполнить свою работу. Как правило, если к этой книге прилагается пример кода, вы можете использовать его в своих программах и документации. Вам не нужно обращаться к нам за разрешением, если вы не воспроизводите значительную часть кода. Например, для написания программы, использующей несколько фрагментов кода из этой книги, разрешения не требуется. Для продажи или распространения примеров из книг O'Reilly требуется разрешение. Чтобы ответить на вопрос, цитируя эту книгу и код примера, разрешения не требуется. Включение значительного количества примеров кода из этой книги в документацию по вашему продукту требует разрешения.

Мы приветствуем указание авторства при цитировании, но обычно не настаиваем на нем. Ссылка при цитировании обычно включает название, автора, издателя и ISBN. Например: "Карманный справочник по конвейерам данных" Джеймса Денсмора (O'Reilly). Copyright 2021 Джеймс Денсмор, 978-1-492-08783-0".

Если вы считаете, что использование примеров программного кода выходит за рамки добросовестного или нарушает разрешения, данные выше, свяжитесь с нами по адресу:

permissions@oreilly.com.

Благодарности

Спасибо всем сотрудникам O'Reilly, которые помогли осуществить издание этой книги, особенно Джессике Хаберман и Корбину Коллинзу. Бесценные отзывы трех замечательных технических рецензентов, Джой Пэйтон, Гордона Вонга и Скотта Хейнса, помогли внести существенные улучшения. Наконец, спасибо моей жене Аманде за ее поддержку с того момента, как возникла идея этой книги, а также моей собаке Иззи за то, что сидела рядом со мной в течение бесчисленных часов, проведенных за работой.

ГЛАВА 1

Введение в конвейеры данных

За каждой красивой приборной панелью, моделью машинного обучения и идеями, способными изменить бизнес, стоят данные. Не просто необработанные данные, а данные, собранные из многочисленных источников, которые необходимо очистить, обработать и объединить для получения ценной информации. Знаменитая фраза "данные — это новая нефть" оказалась удивительно верной. Как и в случае с нефтью, ценность данных заключается в их потенциале после того, как они будут очищены и доставлены потребителю. Как и в случае с нефтью, для передачи данных на каждом этапе цепочки создания стоимости требуются эффективные конвейеры.

В этом карманном справочнике обсуждается вопрос, что представляют собой конвейеры данных, и показано, как они вписываются в современную экосистему данных. В нем рассмотрены общие соображения и ключевые моменты принятия решений при реализации конвейеров, такие как пакетный или потоковый прием данных, разработка или покупка инструментов и многое другое. В справочнике приведены примеры наиболее распространенных решений, принимаемых специалистами по данным, а также обсуждаются основополагающие концепции, применимые к собственным решениям, платформам с открытым исходным кодом и коммерческим продуктам.

Что такое конвейеры данных?

Конвейеры данных (data pipeline) — это совокупность процессов, которые преобразовывают данные из различных источников и перемещают их в место назначения, где они могут приобрести новую ценность. Конвейеры лежат в основе аналитики, отчетности и машинного обучения.

Сложность конвейера данных зависит от размера, состояния и структуры исходных данных, а также от потребностей аналитического проекта. В своей простейшей форме конвейеры могут извлекать данные только из одного источника, например REST API, и загружать их в единственное место назначения, например в таблицу SQL в хранилище данных. Однако на практике конвейеры обычно состоят из нескольких этапов, куда входят извлечение данных, их предварительная обработка, проверка, а иногда и обучение или запуск модели машинного обучения перед доставкой данных в пункт назначения. Конвейеры часто содержат задачи из нескольких систем и языков программирования. Более того, специалисты по обработке данных обычно располагают многочисленными конвейерами данных, которые имеют общие зависимости и должны координироваться. На рис. 1.1 показан простой конвейер данных.

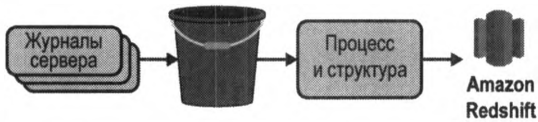

Рис. 1.1. Простой конвейер, который загружает данные журнала сервера в корзину S3, выполняет некоторую базовую обработку и структурирование и передает результаты в БД Amazon Redshift

Кто строит конвейеры данных?

С ростом популярности облачных вычислений и программного обеспечения как услуги (Software as a Service, SaaS) резко возросло количество источников данных, которые организациям приходится использовать. В то же время спрос на данные для моделей машинного обучения, исследований в области науки о данных и идей, нуждающихся в неотложной реализации, сегодня выше, чем когда-либо. *Инженерия данных* приобрела ключевую роль в командах аналитиков, стремящихся не отстать от жизни. *Инженеры данных* специализируются на создании и обслуживании конвейеров данных, лежащих в основе аналитической экосистемы.

Предназначение инженера данных — не просто загружать данные в хранилище. Инженеры данных тесно сотрудничают с уче-

ными и аналитиками, чтобы понять, какие манипуляции они собираются выполнять с данными, и помочь реализовать их потребности в масштабе производства.

Инженеры данных гордятся тем, что обеспечивают достоверность и своевременность данных, которые они предоставляют. Это означает тестирование, оповещение и создание планов на случай непредвиденных обстоятельств, когда что-то пойдет не так. И да, рано или поздно обязательно что-то пойдет не так!

Конкретные навыки инженера данных в некоторой степени зависят от стека технологий, который использует его организация. Однако есть некоторые общие навыки, присущие всем хорошим инженерам данных.

Основы SQL и хранилища данных

Инженеры данных должны знать, как делать запросы к базам данных, а SQL, как известно, является универсальным языком для этого. Опытные инженеры данных умеют писать высокопроизводительные SQL-запросы и понимают основы хранения и моделирования данных. Даже если в команду входят специалисты по хранению данных, инженер данных, обладающий навыками работы с хранилищами, является лучшим партнером и может заполнить возникающие технические пробелы.

Python и/или Java

Язык программирования, которым владеет инженер данных, будет зависеть от технологического стека его команды, но в любом случае инженер данных не сможет выполнить всю работу с помощью инструментов *no code* ("без кода"), даже если в его арсенале есть хорошие инструменты. В настоящее время в инжиниринге данных безусловно доминируют Python и Java, но появляются и новички, такие как язык Go.

Распределенные вычисления

Необходимость решать проблемы, связанные с большими объемами данных, и стремление быстро обрабатывать данные, привели к тому, что инженеры данных начали работать с платформами распределенных вычислений. Распределенные вычисления

сочетают в себе мощь нескольких систем для эффективного хранения, обработки и анализа больших объемов данных.

Один из популярных примеров распределенных вычислений в аналитике — экосистема Hadoop, которая включает в себя распределенное хранилище файлов, основанное на распределенной файловой системе Hadoop (HDFS), обработку данных с помощью MapReduce, анализ данных с помощью Pig и многое другое. Apache Spark — еще одна широко известная среда распределенной обработки, которая начинает превосходить по популярности Hadoop.

Хотя не все конвейеры данных нуждаются в распределенных вычислениях, инженеры данных должны знать, как и когда использовать такую структуру.

Основы системного администрирования

Ожидается, что инженер данных будет владеть командной строкой Linux и выполнять такие задачи, как анализ журналов приложений, планирование заданий cron, устранение неполадок брандмауэра и настройка параметров безопасности. Даже при работе с поставщиком облачных услуг, таким как AWS, Azure или Google Cloud, инженер в конечном итоге будет применять эти навыки для организации взаимодействия облачных сервисов и развертывания конвейеров данных.

Понимание общих целей

Хороший инженер данных не просто обладает техническими навыками. Он регулярно общается с заинтересованными сторонами в лице аналитиков и специалистов по данным в своей команде. Инженер данных будет принимать лучшие решения, если одним из первых узнает, в чем заключается конечная цель создания конвейера.

Зачем создавать конвейеры данных?

Точно так же, как верхушка айсберга — это все, что может увидеть проплывающий мимо корабль, конечный продукт рабочего процесса аналитики — это все, что видит бóльшая часть органи-

зации. Руководители видят информационные панели и красивые диаграммы. Маркетинг делится четко упакованными идеями в социальных сетях. Служба поддержки клиентов оптимизирует обеспечение кол-центра персоналом на основе результатов прогнозной модели спроса.

Большинству людей, не связанных с аналитикой, часто невдомек, что за видимой частью скрывается сложный скрытый механизм. Для каждой информационной панели и представления, которые создает аналитик данных, и для каждой прогностической модели, разработанной специалистом по данным, существуют конвейеры данных, работающие за кулисами. Нередко одна информационная панель или даже одна метрика получаются в результате обработки данных, поступающих из нескольких исходных систем. Кроме того, конвейеры данных делают больше, чем просто извлекают данные из источников и загружают их в таблицы БД или простые файлы для использования аналитиками. Необработанные данные очищаются, структурируются, нормализуются, комбинируются, агрегируются, а иногда и анонимизируются или иным образом защищаются. Другими словами, под "ватерлинией" происходит гораздо больше, чем на виду.

Предоставление данных аналитикам и специалистам по данным

Не полагайтесь на аналитиков данных и специалистов по данным, которые самостоятельно ищут и приобретают данные для каждого проекта, встречающегося на их пути. Слишком велики риски принятия решений на основе устаревших данных, неоднозначности доверенных источников и увязания аналитиков в сборе данных. Конвейеры данных гарантируют доставку нужных данных, чтобы остальная часть команды аналитиков могла сосредоточить свои усилия на том, что у них получается лучше всего: на предоставлении информации целевому потребителю.

Как строятся конвейеры?

Наряду с профессией инженера данных в последние годы появилось множество инструментов для создания и поддержки конвейеров данных. Одни из них с открытым исходным кодом, другие коммерческие, а есть и такие, которые разработаны и применяются локально. Некоторые конвейеры написаны на Python, некоторые — на Java, на каком-то другом языке или вообще не содержат кода.

В этом карманном справочнике представлено исследование некоторых наиболее популярных продуктов и сред для построения конвейеров, а также обсуждение того, как определить наиболее подходящие инструменты и решения исходя из потребностей и ограничений вашей организации.

Хотя в книге нет подробных описаний всех таких продуктов, для некоторых из них приведены примеры и образцы кода. Весь программный код в этой книге написан на Python и SQL. Это наиболее распространенные и, пожалуй, самые доступные языки для построения конвейеров данных.

Кроме того, конвейеры не только строят — их нужно контролировать, обслуживать и развивать. Перед инженерами данных стоит задача не просто организовать доставку данных тем или иным способом, но и создать конвейеры и поддерживающую инфраструктуру, которые обеспечивают надежную, безопасную и своевременную доставку и обработку данных. Это нелегкий подвиг, но когда все сделано правильно, можно спокойно сосредоточиться на извлечении ценной информации из данных организации.

ГЛАВА 2

Современная инфраструктура данных

Прежде чем принимать решение о составе и устройстве будущего конвейера, нужно хорошо понимать, из чего состоит современный стек данных. Как и в большинстве случаев в сфере технологий, не существует единого правильного способа проектирования вашей аналитической экосистемы или выбора продуктов и поставщиков. Тем не менее есть некоторые базовые потребности и концепции, которые стали отраслевыми стандартами и заложили основу для современных подходов к реализации конвейеров.

Давайте взглянем на ключевые компоненты такой инфраструктуры, показанные на рис. 2.1. В следующих главах будет подробно рассказано, как каждый компонент влияет на проектирование и реализацию конвейеров данных.

Рис. 2.1. Ключевые компоненты современной инфраструктуры данных

Разнообразие источников данных

Большинство организаций имеют десятки, если не сотни, источников данных, поставляющих рабочий материал командам аналитиков. Источники данных различаются по многим параметрам, рассматриваемым в этом разделе.

Принадлежность исходной системы

Группа аналитиков обычно получает данные из исходных систем, созданных и принадлежащих организации, а также из сторонних инструментов и поставщиков. Например, компания электронной коммерции может хранить данные из корзины покупателя в БД PostgreSQL (также известной как Postgres), связанной с веб-приложением магазина. Она также может использовать сторонний инструмент веб-аналитики, такой как Google Analytics, для отслеживания функционирования своего веб-сайта. Комбинация двух источников данных (показана на рис. 2.2) необходима для полного понимания поведения клиентов, ведущего к покупке. Таким образом, конвейер данных, который заканчивается анализом поведения покупателя, начинается со сбора данных из обоих источников.

ПРИМЕЧАНИЕ

Под *сбором данных* (data ingestion) подразумевается извлечение данных из источника и их загрузка в приемник, с приведением к надлежащему виду по мере необходимости.

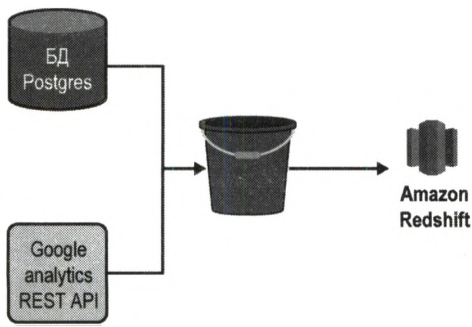

Рис. 2.2. Простой конвейер с данными из нескольких источников, загружаемыми в корзину S3, а затем в БД Redshift

Понимание прав собственности на исходные системы важно по нескольким причинам. При использовании сторонних источников данных вы, вероятно, ограничены в отношении того, к каким данным и каким образом вы можете получить доступ. Большинство поставщиков предоставляют REST API, но немногие обеспечат вам прямой доступ к вашим данным в виде БД SQL, и лишь некоторые из них дадут вам возможность настройки избирательного доступа к данным и уровня их детализации.

Внутренние системы предоставляют группе аналитиков больше возможностей для настройки доступа к данным, а также выбора методов доступа. Однако они создают и проблемы. Были ли системы построены с учетом сбора данных? Часто ответ отрицательный, что влечет за собой различные последствия: от перегрузки системы в процессе сбора данных до невозможности их постепенной загрузки. Если вам повезет, у инженерной группы, владеющей исходной системой, будет время и желание работать с вами, но в условиях ограниченности ресурсов вы можете обнаружить, что обладание собственной системой не лучше работы с внешним поставщиком.

Интерфейс сбора и структура данных

Независимо от того, кому принадлежат исходные данные, как вы их получаете и в какой форме — это первое, что проверит инженер данных при создании нового канала сбора данных. Прежде всего, каков интерфейс доступа к данным? В число наиболее распространенных интерфейсов входят следующие:

- база данных, стоящая за приложением, например БД Postgres или MySQL;
- уровень абстракции поверх системы, такой как REST API;
- платформа обработки потоков, такая как Apache Kafka;
- общая сетевая файловая система или сегмент облачного хранилища, содержащий журналы, файлы со значениями, разделенными запятыми (CSV), и другие простые файлы;
- хранилище данных, или "озеро" данных;
- данные в БД HDFS или HBase.

Помимо разных интерфейсов существуют и различные структуры данных. Вот несколько распространенных примеров:

- JSON из REST API;
- хорошо структурированные данные из БД MySQL;
- JSON в столбцах таблицы БД MySQL;
- полуструктурированные данные журнала;
- CSV, формат фиксированной ширины (FWF) и другие форматы плоских файлов;
- JSON в неструктурированных файлах;
- потоковый вывод из Kafka.

Каждому интерфейсу и структуре данных присущи свои достоинства и недостатки. С хорошо структурированными данными часто проще всего работать, но обычно они структурированы в интересах конкретного приложения или веб-сайта. Помимо сбора данных, вероятно, потребуются дальнейшие шаги в конвейере для их очистки и преобразования в структуру, более подходящую для аналитического проекта.

Полуструктурированные данные, такие как JSON, становятся все более распространенными и имеют все преимущества структуры пар "атрибут – значение" и вложенности объектов. Однако, в отличие от реляционной БД, нет гарантии, что каждый объект в одном и том же наборе данных будет иметь одинаковую структуру. Как вы увидите далее в этой книге, технология работы с отсутствующими или неполными данными в конвейере зависит от контекста и становится все более необходимой по мере снижения жесткости структуры данных.

Для некоторых аналитических задач характерны неструктурированные данные. Например, модели обработки естественного языка (Natural Language Processing, NLP) нуждаются в огромных объемах произвольных текстовых данных для обучения и проверки. Для проектов компьютерного зрения (Computer Vision, CV) требуются изображения и видеоконтент. Даже менее сложные проекты, такие как сбор данных с веб-страниц, нуждаются в бесплатных текстовых данных из Интернета в дополнение к полуструктурированной HTML-разметке веб-страницы.

Объем данных

Хотя инженеры по обработке данных и менеджеры по найму персонала любят хвастаться петабайтными наборами данных, реальность такова, что большинство организаций ценят маленькие наборы данных не меньше, чем большие. Кроме того, принято одновременно собирать и моделировать наборы данных разного объема. Хотя проектные решения на каждом этапе конвейера должны учитывать объем данных, большой объем не означает большую ценность.

Все это говорит о том, что у большинства организаций есть хотя бы один набор данных большого объема, который является ключевым для нужд анализа. Но что такое большой объем? Простого определения не существует, а что касается конвейеров, то лучше ориентироваться на непрерывный спектр объемов, а не разделять данные на наборы большого и малого объема.

> **ПРИМЕЧАНИЕ**
>
> На протяжении всей этой книги вы увидите, что чрезмерное упрощение процедур сбора и обработки данных, приводящее к длительным и неэффективным вычислительным процессам, таит в себе столько же опасностей, сколько и проектирование чрезмерно сложных конвейерных этапов, когда объем данных или сложность задачи невелики.

Чистота и достоверность данных

Наряду с обширным разнообразием источников существуют и большие различия в качестве исходных данных. Как гласит популярная поговорка аналитиков, "мусор на входе, мусор на выходе". Важно понимать ограничения и недостатки исходных данных и устранять их в соответствующих разделах ваших конвейеров.

Есть много общих признаков "беспорядочных данных", в том числе следующие:

- повторяющиеся или неоднозначные записи;
- потерянные записи;
- неполные или отсутствующие записи;
- ошибки кодирования текста;

- несовместимые форматы (например, телефонные номера с дефисами или без них);
- неправильно помеченные или неразмеченные данные

Конечно, есть множество других признаков, а также вопросов к достоверности данных, характерных для контекста исходной системы.

Не существует волшебного средства для обеспечения чистоты и достоверности данных, но в современной экосистеме данных применяются ключевые характеристики и подходы, которые будут рассмотрены в этой книге:

- Предполагай худшее, надейся на лучшее.

  Идеальные первичные наборы данных существуют только в академической литературе. Исходите из предположения, что ваши входные наборы данных будут содержать многочисленные проблемы с достоверностью и согласованностью, и создавайте конвейеры, которые идентифицируют и очищают данные для получения правильного результата на выходе.

- Очищайте и проверяйте данные в системе, которая лучше всего подходит для этого.

  Бывают случаи, когда лучше отложить очистку данных до более поздней стадии конвейера. Например, современные конвейеры обычно используют подход *извлечения-загрузки-преобразования* (Extract-Load-Transform, ELT), а не *извлечения-преобразования-загрузки* (Extract-Transform-Load, ETL) для хранения данных (подробнее в *главе 3*). Иногда бывает выгоднее загружать данные в озеро данных в практически необработанном виде и выполнять структурирование и очистку позже в конвейере. Другими словами, выбирайте подходящий инструмент для правильной работы и не спешите с процессами очистки и проверки.

- Проверяйте данные часто.

  Даже если вы не очищаете данные в начале конвейера, не ждите окончания конвейера, чтобы проверить их. Вам будет гораздо труднее определить, где что-то пошло не так. И наоборот, не ограничивайтесь проверкой один раз в начале конвейера в надежде, что на последующих этапах все пойдет хорошо. Проверка данных будет рассмотрена в *главе 8*.

Задержка и пропускная способность исходной системы

Необходимость часто извлекать большие объемы данных из исходных систем — распространенный вариант использования современного стека данных. Однако это сопряжено с трудностями. Шаги извлечения данных в конвейерах должны соответствовать ограничениям скорости API, тайм-аутам соединения, медленным загрузкам и требованиям владельцев исходных систем, которые недовольны нагрузкой на свои системы.

> **ПРИМЕЧАНИЕ**
>
> Как будет подробно рассказано в *главах 4* и *5*, сбор данных — это первый шаг в большинстве конвейеров данных. Поэтому выяснение характеристик исходных систем и их данных является первым шагом в проектировании конвейеров и принятии решений относительно дальнейшей инфраструктуры.

Облачные хранилища данных и озера данных

Три вещи изменили ландшафт аналитики и хранилищ данных за последние 10 лет, и все они связаны с появлением крупных поставщиков общедоступных облачных сервисов (Amazon, Google и Microsoft):

- простота создания и развертывания конвейеров данных, озер данных, хранилищ и выполнения аналитики в облаке. Больше не нужно ждать милости от ИТ-отделов и утверждения бюджета для крупных первоначальных затрат. Управляемые услуги, в частности базы данных, приобрели огромную популярность;
- постоянное снижение расходов на хранение данных в облаке;
- появление хорошо масштабируемых столбцовых БД, таких как Amazon Redshift, Snowflake и Google Big Query.

Эти изменения вдохнули новую жизнь в хранилища данных и породили концепцию озера данных. Хотя в *главе 5* хранилища и озера данных рассматриваются более подробно, сейчас стоит

дать краткое определение и того и другого, чтобы прояснить их место в современной экосистеме данных.

Хранилище данных — это БД, в которой данные из разных систем хранятся и моделируются для поддержки анализа и других действий, связанных с ответами на вопросы. Данные в хранилище структурированы и оптимизированы для запросов отчетов и анализа.

Озеро данных — это тоже место хранения данных, но без структуры или оптимизации запросов, присущих хранилищу данных. Скорее всего, озеро будет содержать большой объем данных различных типов. Например, одно озеро данных может содержать набор сообщений в блогах в виде текстовых файлов, плоские файлы выгрузок из реляционной БД и объекты JSON, содержащие события, сгенерированные датчиками в промышленной системе. Озеро может даже хранить структурированные данные, как стандартная БД, хотя и не оптимизировано для запроса таких данных для целей отчетности и анализа.

В одной и той же экосистеме могут присутствовать как хранилища, так и озера данных, и конвейеры часто перемещают данные между ними.

Инструменты сбора данных

Необходимость переноса данных из одной системы в другую характерна почти для всех конвейеров данных. Как было сказано ранее в этой главе, специалистам по работе с данными приходится иметь дело с разнообразными источниками, из которых они могут извлекать данные. К счастью, в современной инфраструктуре данных доступен ряд коммерческих продуктов и инструментов с открытым исходным кодом.

В этом карманном справочнике я расскажу о некоторых наиболее распространенных инструментах и фреймворках, в том числе:

- Singer;
- Stitch;
- Fivetran.

Несмотря на распространенность этих инструментов, некоторые команды разработчиков предпочитают создавать собственный

код для сбора данных. Некоторые даже разрабатывают собственные фреймворки. Причины различаются в зависимости от организации, но часто связаны со стоимостью, сложившейся культурой разработки вместо покупки и опасениями по поводу юридических рисков и рисков безопасности, связанных с доверием к внешнему поставщику. В *главе 5* обсуждаются компромиссы между разработкой и покупкой, которые присущи только инструментам сбора данных. Особый интерес представляет ответ на вопрос о том, чем обоснован выбор коммерческого решения — чтобы инженерам данных было проще встраивать сбор данных в свои конвейеры, или чтобы дать возможность инженерам, не занимающимся "сырыми" данными (таким как аналитики данных), создавать точки сбора данных самостоятельно.

Как будет показано в *главах 4* и *5*, прием данных традиционно соответствует этапам *извлечения* и *загрузки* процессов ETL или ELT. Некоторые инструменты сосредоточены только на этих шагах, в то время как другие также предоставляют пользователю определенные возможности преобразования. На практике я обнаружил, что большинство специалистов по обработке данных предпочитают ограничивать количество преобразований, которые они выполняют во время сбора данных, и поэтому выбирают инструменты, которые хороши в двух вещах: извлечении данных из источника и загрузке их в место назначения.

Инструменты преобразования и моделирования данных

Хотя основная часть этой главы посвящена перемещению данных между источниками и получателями (сбор данных), конвейеры данных выполняют гораздо больше операций, чем простое перемещение данных. Конвейеры также состоят из задач, которые преобразуют и моделируют данные для новых целей, таких как машинное обучение, анализ и отчетность.

Термины *моделирование данных* (data modelling) и *преобразование данных* (data transformation) часто взаимозаменяемы, но в этой книге я буду различать их:

- *Преобразование данных* — это широкое понятие, которому соответствует буква *T* от слова Transformation в аббревиатурах

ETL или ELT. Преобразование может быть очень простым, например преобразование метки времени, хранящейся в таблице, из одного часового пояса в другой. Это также может быть сложная операция, создающая новую метрику из нескольких исходных столбцов, которые агрегируются и фильтруются с помощью определенной бизнес-логики.

- *Моделирование данных* — это более конкретный тип преобразования данных. Модель структурирует и определяет данные в формате, понятном и оптимизированном для анализа. Модель данных обычно представлена в виде одной или нескольких таблиц в хранилище данных. Процесс создания моделей данных обсуждается более подробно в *главе 6*.

Как и при сборе данных, в современной инфраструктуре данных применяется ряд методологий и инструментов. Как отмечалось ранее, некоторые инструменты сбора данных предоставляют определенные возможности преобразования, но зачастую их выбор весьма ограничен. Например, для защиты *информации, позволяющей установить личность* (Personally Identifiable Information, PII), может потребоваться преобразовать адрес электронной почты в хешированное значение, которое хранится в конечном пункте конвейера. Такое преобразование обычно выполняется в процессе сбора данных.

Для более сложных преобразований и моделирования данных я предпочитаю искать инструменты и среды, специально разработанные под конкретную задачу, такие как dbt (см. *главу 9*). Кроме того, преобразование данных часто зависит от контекста и может быть реализовано на языке, знакомом инженерам и аналитикам данных, например SQL или Python.

Модели данных, которые будут использоваться для анализа и составления отчетов, обычно определяются и записываются на языке SQL или с помощью графических пользовательских интерфейсов. Так же как и при выборе между разработкой и покупкой, при построении моделей на основе SQL необходимо учитывать возможность использования *инструментов без кода* (no-code tool). SQL — это очень доступный язык, общий как для инженеров, так и для аналитиков данных. Это позволяет аналитику работать непосредственно с данными и оптимизировать структуру моделей для своих нужд. Он также применяется почти в каж-

дой организации, таким образом обеспечивая знакомую точку входа для новых сотрудников. В большинстве случаев желателен выбор платформы преобразования, которая поддерживает построение моделей данных в SQL, а не через визуальный пользовательский интерфейс. Вы получите гораздо больше возможностей для настройки и будете полностью контролировать процесс разработки.

В *главе 6* подробно обсуждаются преобразование и моделирование данных.

Платформы для оркестровки рабочих процессов

По мере роста сложности и количества конвейеров данных в организации важно своевременно внедрить *платформу оркестровки рабочих процессов* (Workflow Orchestration Platform) в инфраструктуру данных. Эти платформы управляют планированием и потоком задач в конвейере. Представьте себе конвейер с дюжиной задач, начиная от приложения сбора данных, написанного на Python, и заканчивая преобразованиями данных, написанными на SQL, которые должны выполняться в определенной последовательности в течение дня. Планирование и управление зависимостями между каждой задачей — непростая работа. Каждая команда специалистов по данным сталкивается с этой проблемой, но, к счастью, существует множество платформ для оркестровки рабочих процессов, принимающих на себя значительную часть работы.

> **ПРИМЕЧАНИЕ**
>
> Платформы оркестровки рабочих процессов также называются *системами управления рабочими процессами* (Workflow Management Systems, WMS), *платформами оркестровки* или *средами оркестровки*. Я употребляю эти термины взаимозаменяемо.

Некоторые платформы, такие как Apache Airflow, Luigi и AWS Glue, предназначены для более общих случаев использования и поэтому пригодны для самых разных конвейеров данных. Другие, такие как Kubeflow Pipelines, предназначены для более конкретных случаев и платформ (рабочие процессы машинного обу-

чения, построенные на контейнерах Docker в случае Kubeflow Pipelines).

Направленные ациклические графы (DAG)

Почти все современные платформы оркестровки представляют поток и зависимости задач в конвейере в виде графа. Однако конвейерные графы имеют некоторые специфические ограничения.

Этапы конвейера всегда *направлены*, т. е. они начинаются с общей задачи или нескольких задач и заканчиваются конкретной задачей или задачами. Это необходимо, чтобы гарантировать путь выполнения. Другими словами, направленность гарантирует, что новые задачи не будут запущены до тех пор, пока не завершены (причем успешно) все предшествующие задачи.

Конвейерные графы также должны быть ациклическими, т. е. задача не может указывать на ранее выполненную задачу. Другими словами, данные не могут вернуться назад. В противном случае конвейер мог бы работать бесконечно!

Исходя из этих двух ограничений, конвейеры оркестровки создают так называемые *направленные ациклические графы* (Directed Acyclic Graph, DAG). На рис. 2.3 показан простой DAG. В этом примере задача *A* должна быть завершена до того, как можно будет начать задачи *B* и *C*. Как только они обе будут выполнены, можно приступить к задаче *D*. Как только задача *D* завершена, конвейер останавливается.

DAG — это *представление* набора задач, а не место, где определяется их логика. Платформа оркестровки способна выполнять задачи всех видов.

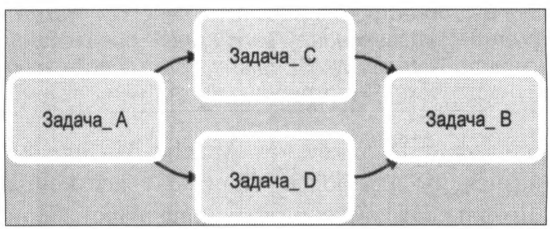

Рис. 2.3. DAG с четырьмя задачами. После завершения задачи *A* запускаются задачи *B* и *C*. Когда они обе завершатся, запускается задача *D*

Например, рассмотрим конвейер данных с тремя задачами. На рис. 2.4 он представлен в виде DAG.

- Первая задача выполняет сценарий SQL, который запрашивает данные из реляционной БД и сохраняет результат в файле CSV.
- Вторая задача запускает сценарий Python, который загружает CSV-файл, очищает данные и затем изменяет их форму перед сохранением новой версии файла.
- Наконец, третья задача запускает команду `COPY` в SQL, загружает файл CSV, созданный второй задачей, в хранилище данных Snowflake.

Рис. 2.4. DAG с тремя последовательно выполняемыми задачами для извлечения данных из БД SQL, очистки и изменения формы данных с помощью сценария Python, а затем загрузки полученных данных в хранилище

Платформа оркестровки запускает каждую задачу, но логика задач существует в виде кода SQL и Python, который выполняется в разных системах в рамках инфраструктуры данных.

В *главе 7* более подробно обсуждаются платформы для оркестровки рабочих процессов и приводятся практические примеры оркестровки конвейера в Apache Airflow.

Настройка вашей инфраструктуры данных

Редко можно встретить две организации с одинаковой инфраструктурой данных. Как правило, организации выбирают инструменты и поставщиков, отвечающих их конкретным потребностям, а все остальное создают самостоятельно. В этой книге я подробно рассказываю о некоторых наиболее популярных инструментах и продуктах, но их выбор становится шире с каждым годом.

Как отмечалось ранее, в зависимости от культуры и ресурсов вашей организации вам может быть предложено построить большую часть инфраструктуры данных самостоятельно или вместо

этого полагаться на поставщиков SaaS. Независимо от выбора между разработкой и покупкой, вам вполне по силам построить высококачественную инфраструктуру данных, необходимую для создания столь же высококачественных конвейеров данных.

Важно понимать ваши ограничения (финансовые и инженерные ресурсы, безопасность и устойчивость к юридическим рискам) и вытекающие из этого компромиссы. Я говорю об этом на протяжении всей книги и указываю на ключевые моменты принятия решений при выборе продукта или инструмента.

Стандартные шаблоны конвейеров данных

Даже для опытных инженеров данных проектирование нового конвейера данных — это каждый раз новое приключение. Как обсуждалось в *главе 2*, различные источники данных и инфраструктура создают как проблемы, так и возможности. Кроме того, конвейеры строятся с разными целями и ограничениями. Должны ли данные обрабатываться практически в режиме реального времени? Можно ли обновлять их ежедневно? Данные будут отображаться на информационной панели или поступать на вход модели машинного обучения?

К счастью, для конвейеров данных есть несколько стандартных шаблонов, которые доказали свою эффективность и могут быть доработаны для многих вариантов использования. В настоящей главе я дам определение этим шаблонам. В последующих главах реализуются конвейеры, построенные на их основе.

Шаблоны ETL и ELT

Возможно, самые известные шаблоны — это ETL и его более современный брат ELT. Оба шаблона широко применяются в хранилищах данных и бизнес-аналитике. В последние годы они служат основой конвейерных шаблонов для моделей науки о данных и машинного обучения, работающих в производственной среде. Они настолько хорошо известны, что многие люди используют эти термины как синонимы конвейеров данных, а не шаблонов, по которым построены различные конвейеры.

Поскольку эти шаблоны уходят корнями в хранилища данных, проще всего рассматривать их в контексте хранилищ. В следующих разделах главы будут описаны конкретные случаи использования.

Оба шаблона представляют собой подходы к обработке данных, которые служат для передачи данных в хранилище и делают их полезными для аналитиков и инструментов отчетности. Разница между шаблонами заключается в порядке выполнения их последних двух этапов (преобразование и загрузка), но последствия выбора между ними весьма существенны, как я объясню в этой главе. Начнем со знакомства с этапами ETL и ELT.

На этапе *извлечения* (extract) данные собираются из различных источников для подготовки к загрузке и преобразованию. В *главе 2* обсуждалось разнообразие этих источников и методов извлечения.

На этапе *загрузки* (load) в конечный пункт назначения передаются либо необработанные данные (при ELT), либо полностью преобразованные данные (при ETL). В любом случае конечный результат — загрузка данных в хранилище, озеро данных или другое место назначения.

На этапе *преобразования* (transform) необработанные данные из каждой исходной системы объединяются и форматируются таким образом, чтобы они были полезны для аналитиков, инструментов визуализации или любого другого варианта использования, для которого предназначен ваш конвейер. На этом шаге происходит много разных действий, независимо от того, разрабатываете ли вы свой процесс как ETL или ELT; все они подробно рассматриваются в *главе 6*.

Разделение извлечения и загрузки

Сочетание этапов извлечения и загрузки часто называют *сбором данных* (data ingestion). Возможности извлечения и загрузки часто тесно связаны и упакованы вместе в программных средах. Особенно это характерно для ELT и подшаблона EtLT (обратите внимание на строчную букву t), который определяется далее в этой главе. Однако при проектировании конвейеров все же лучше рассматривать эти два этапа как отдельные из-за сложности координации процессов извлечения и загрузки в разных системах и инфраструктурах.

В *главах 4* и *5* более подробно описываются методы сбора данных и приводятся примеры реализации с использованием общих платформ.

Преимущество ELT перед ETL

ETL был золотым стандартом шаблонов конвейеров данных на протяжении десятилетий. Хотя он все еще применяется, в последнее время предпочтительным шаблоном стал ELT. Почему? До появления современных хранилищ данных, в основном в облаке (см. *главу 2*), группы специалистов по обработке данных не имели доступа к хранилищам с вычислительными ресурсами, необходимыми для загрузки огромных объемов "сырых" данных и преобразования их в пригодные для использования модели данных в одном месте. Кроме того, хранилищами данных в то время были строковые БД, которые хорошо обрабатывали транзакционные сценарии, но не объемные массовые запросы, ставшие обычным явлением в аналитике. Поэтому данные сначала извлекали из исходных систем, а затем преобразовывали в отдельной системе перед загрузкой в хранилище для окончательного моделирования данных и запросов со стороны аналитиков и инструментов визуализации.

Большинство современных хранилищ построено на основе масштабируемых колоночных БД (иногда их называют столбцовыми. — *Прим. пер.*), которые могут не только хранить большие массивы данных, но и эффективно выполнять массовые преобразования. Многое изменилось благодаря эффективности ввода-вывода колоночных БД, сжатию данных и возможности распределять данные и запросы по множеству узлов, способных работать вместе для обработки данных. Мы сосредоточимся на извлечении данных и их загрузке в хранилище, где затем можно выполнить необходимые преобразования для завершения конвейера.

Влияние разницы между хранилищами данных на основе строк и столбцов невозможно переоценить. На рис. 3.1 приведен пример того, как записи хранятся на диске в БД на основе строк, такой как MySQL или Postgres. Каждая строка БД хранится на диске вместе с остальными, в одном или нескольких блоках в зависимости от размера каждой записи. Если запись меньше одного блока или не делится точно на размер блока, часть дискового пространства остается неиспользованной.

Рассмотрим использование БД *оперативной обработки транзакций* (Online Transaction Processing, OLTP) на примере веб-прило-

жения электронной коммерции, которое задействует БД MySQL. Веб-приложение запрашивает чтение и запись из БД MySQL и в нее, часто выбирая несколько значений из каждой записи, например сведения о заказе на странице подтверждения заказа. Также вероятно, что приложение будет запрашивать или обновлять только один заказ за обращение. Таким образом, хранение на основе строк является оптимальным, поскольку данные, необходимые приложению, хранятся в непосредственной близости на диске, а объем данных, запрашиваемых за один раз, невелик.

OrderId	CustomerId	ShippingCountry	OrderTotal
1	1258	US	55.25
2	5698	AUS	125.36
3	2265	US	776.95
4	8954	CA	32.16

Block 1	1, 1258, US, 55.25
Block 2	2, 5698, AUS, 125.36
Block 3	3, 2265, US, 776.95
Block 4	4, 8954, CA, 32.16

Рис. 3.1. Таблица, хранящаяся в строковой БД. Каждый блок содержит запись (строку) из таблицы

В рассматриваемом случае неэффективное использование дискового пространства из-за того, что записи оставляют пустое место в блоках, является разумным компромиссом, поскольку высокая скорость чтения и записи отдельных строк важнее, чем занимаемое место. Однако в аналитике ситуация обратная. Вместо того, чтобы часто читать и записывать небольшие объемы данных, мы относительно редко читаем и записываем большие объемы данных. Кроме того, маловероятно, что для аналитического запроса потребуются многие или все столбцы таблицы. Скорее всего, понадобится лишь один из множества столбцов таблицы.

Возьмем, к примеру, таблицу заказов в нашем вымышленном приложении электронной коммерции. Среди прочего она содержит стоимость заказа, а также название страны, в которую он будет отправлен. В отличие от веб-приложения, которое работает с заказами по одному, аналитик, использующий хранилище данных, захочет проанализировать большой массив заказов. Кроме

того, таблица, содержащая данные о заказах в хранилище, имеет дополнительные столбцы, содержащие значения из нескольких таблиц в нашей БД MySQL. Например, она может содержать информацию о покупателе, разместившем заказ. Возможно, аналитик хочет суммировать все заказы, сделанные клиентами с активными в данный момент учетными записями. Такой запрос может включать миллионы записей, но считывать их можно только из двух столбцов: `OrderTotal` и `CustomerActive`. В конце концов, аналитика — это не создание или изменение данных (как в OLTP), а вывод метрик и интерпретация данных.

Как показано на рис. 3.2, колоночная БД, такая как Snowflake или Amazon Redshift, хранит данные в блоках диска по столбцам, а не по строкам. В нашем случае запрос, написанный аналитиком, должен получить доступ только к блокам, в которых хранятся значения `OrderTotal` и `CustomerActive`, а не к блокам, в которых хранятся строковые записи, такие как в БД MySQL. Следовательно, требуется меньше дисковых операций ввода-вывода, а также меньше данных для загрузки в память для выполнения операций фильтрации и суммирования, предусмотренных запросом аналитика. Еще одно преимущество — сокращение объема хранилища благодаря тому факту, что блоки могут быть целиком заполнены

OrderId	CustomerId	Shipping Country	Order Total	Customer Active
1	1258	US	55.25	TRUE
2	5698	AUS	125.36	TRUE
3	2265	US	776.95	TRUE
4	8954	CA	32.16	FALSE

Block 1	1, 2, 3, 4
Block 2	1258, 5698, 2265, 8954
Block 3	US, AUS, US, CA
Block 4	55.25, 125.36, 776.95. 32.16
Block 5	TRUE, TRUE, TRUE, FALSE

Рис. 3.2. Таблица, хранящаяся в колоночной БД. Каждый блок диска содержит данные из одного и того же столбца. Заливкой выделены два столбца, задействованные в нашем примере запроса. В данном случае только эти блоки нужны для выполнения запроса. Каждый блок содержит данные одного типа, что делает сжатие оптимальным

и оптимально сжаты, поскольку в каждом блоке хранится один и тот же тип данных, а не несколько типов, которые обычно встречаются в одной записи на основе строк.

В целом благодаря появлению колоночных БД хранение, преобразование и запросы к большим наборам данных становятся намного эффективнее. Инженеры данных могут использовать их в своих интересах, создавая специализированные этапы конвейера, сосредоточенные на извлечении и загрузке данных в хранилища, где они могут быть преобразованы, смоделированы и запрошены аналитиками и специалистами по данным, которым удобнее работать в пределах БД. Вот почему ELT стал идеальным шаблоном для конвейеров хранилища данных, а также других вариантов применения в машинном обучении и разработке продуктов на основе данных.

Подшаблон EtLT

Когда ELT стал доминирующим шаблоном, выяснилось, что некоторые преобразования после извлечения, но перед загрузкой, по-прежнему полезны. Однако в отличие преобразования, включающего бизнес-логику или моделирование данных, этот тип преобразования имеет более ограниченный охват. Я называю это преобразованием *t* (строчная буква) или EtLT.

Вот некоторые примеры преобразований, соответствующих подшаблону EtLT:

- удаление дубликатов записей в таблице;
- разделение параметров URL на отдельные компоненты;
- маскировка или иное сокрытие конфиденциальных данных.

Эти типы преобразований либо полностью отделены от бизнес-логики, либо, как в случае маскировки конфиденциальных данных, иногда должны выполняться как можно раньше в конвейере по юридическим причинам или по соображениям безопасности. Кроме того, важно выбрать подходящий инструмент для правильной работы. Как более подробно показано в *главах 4* и *5*, большинство современных хранилищ загружают данные наиболее эффективно, если они хорошо подготовлены. В конвейерах, перемещающих большие объемы данных, или там, где задержка

является ключевым фактором, выполнение некоторых базовых преобразований между этапами извлечения и загрузки стоит затраченных усилий.

Далее вы можете считать, что остальные ELT-подобные шаблоны, также включают в себя подшаблон EtLT.

ELT в анализе данных

ELT стал наиболее распространенным и, на мой взгляд, самым оптимальным шаблоном для конвейеров, построенных для анализа данных. Как уже говорилось, колоночные БД хорошо подходят для обработки больших объемов данных. Они также предназначены для работы с широкими таблицами, т. е. таблицами с большим количеством столбцов, благодаря тому, что считываются с диска и загружаются в память только данные в столбцах, необходимых в заданном запросе.

Помимо технических навыков, аналитики данных обычно свободно владеют SQL. Благодаря ELT инженеры данных могут сосредоточиться на этапах извлечения и загрузки (сбор данных), в то время как аналитики могут с помощью SQL преобразовывать данные при подготовке отчетов и анализа. Такое четкое разделение работы невозможно в случае шаблона ETL, поскольку инженеры по данным должны поддерживать весь конвейер. Как показано на рис. 3.3, ELT позволяет членам команды по обработке данных сосредоточиться на своих сильных сторонах с меньшим количеством взаимозависимостей и согласований.

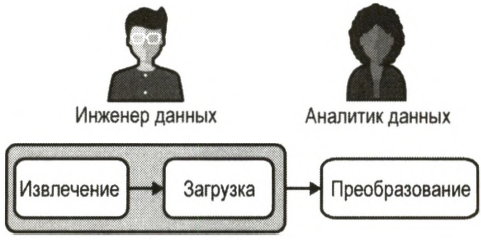

Рис. 3.3. Шаблон ELT позволяет четко разделить обязанности между инженерами данных и аналитиками (или учеными). Каждый специалист может работать автономно с удобными для него инструментами и языками

Кроме того, во время создания процессов извлечения и загрузки шаблон ELT снижает потребность в точном угадывании того, что аналитики будут делать с данными. Хотя для извлечения и загрузки нужных данных требуется хотя бы общее понимание способа их дальнейшего использования, более поздний шаг преобразования дает аналитикам больше возможностей и гибкости.

ПРИМЕЧАНИЕ

С появлением ELT аналитики данных стали более автономными и получили возможность извлекать пользу из данных, не нуждаясь в постоянном взаимодействии с инженерами. Инженеры данных могут сосредоточиться на сборе данных и поддержке инфраструктуры, которая позволяет аналитикам писать и развертывать собственный код преобразования, написанный на языке SQL. Вслед за этими полномочиями появились новые должности, такие как инженер-аналитик. В *главе 6* обсуждается, как аналитики данных и инженеры-аналитики преобразуют данные для построения моделей.

ELT в науке о данных

Конвейеры, созданные для команд *исследователей данных* (data scientist), во многом аналогичны конвейерам, предназначенным для аналитиков. Как и в случае подготовки данных для анализа, инженеры данных сосредоточены на приеме данных в хранилище или озеро данных. Однако потребности исследователей и аналитиков данных различны.

Хотя наука о данных представляет собой обширную область, в целом исследователям данных (в отличие от аналитиков) требуется доступ к более детализированным, а иногда и необработанным данным. В то время как аналитики разрабатывают модели данных, которые формируют метрики и мощные информационные панели (data-dashboard), исследователи проводят дни в изучении данных и построении прогностических моделей. Хотя анализ различий между исследователями и аналитиками данных выходит за рамки этой книги, нужно отметить, что такое различие необходимо учитывать при проектировании конвейеров, предназначенных для исследователей.

Если вы строите конвейеры для исследователей данных, то обнаружите, что этапы извлечения и загрузки шаблона ELT останутся

почти такими же, как и для аналитиков. В *главах 4* и *5* подробно описываются эти шаги. Исследователям данных также иногда бывает полезно работать с некоторыми моделями данных, созданными для аналитиков на этапе преобразования конвейера ELT (*глава 6*), но они, скорее всего, предпочтут ответвление конвейера и будут использовать большую часть данных, полученных во время извлечения-загрузки.

ELT для информационных продуктов и машинного обучения

Данные необходимы не только для анализа, составления отчетов и прогнозных моделей. Они также применяются для создания *информационных продуктов* на основе данных. Вот некоторые распространенные примеры информационных продуктов:

- механизм рекомендаций контента, связанный с главным экраном приложения потокового видео;
- персонализированная поисковая система на веб-сайте электронной коммерции;
- приложение, которое выполняет анализ настроений пользователей на основе отзывов о ресторанах.

Каждый из этих информационных продуктов наверняка будет основан на одной или нескольких моделях *машинного обучения* (Machine Learning, ML), которые нуждаются в данных для обучения и проверки. Такие данные могут поступать из различных исходных систем и подвергаться определенному уровню преобразования для подготовки к использованию в модели. ELT-подобный шаблон хорошо подходит для таких нужд, хотя на всех этапах конвейера, разработанного для информационных продуктов, существует ряд специфических проблем.

Этапы конвейера для машинного обучения

Подобно конвейерам для анализа, которым в первую очередь посвящена эта книга, конвейеры, созданные для машинного обучения, следуют схеме, аналогичной ELT, — по крайней мере, в начале. Разница в том, что вместо этапа, направленного на пре-

образование данных в модели, после извлечения и загрузки данных в хранилище или озеро следуют несколько этапов, связанных с построением и обновлением модели машинного обучения.

Если вы знакомы с основами машинного обучения, эти этапы также могут показаться вам знакомыми:

- *Сбор данных* — на этом этапе реализован процесс, который описан в *главах 4* и *5*. Хотя данные, которые вы собираете, могут быть разными, логика остается в основном одинаковой для конвейеров, созданных как для аналитики, так и для машинного обучения, но с одним дополнительным замечанием для конвейеров машинного обучения. Для набора данных должна быть указана версия таким образом, чтобы модели ML могли позже ссылаться на конкретный набор данных для обучения или проверки. Существует ряд инструментов и подходов для управления версиями наборов данных. Заинтересованным читателям я предлагаю обратиться к *разделу "Дополнительная литература по конвейерам ML"* в конце этой главы.
- *Предварительная обработка данных* — это процесс, где собранные данные очищаются и иным образом подготавливаются для моделей. Например, это этап конвейера, на котором текст токенизируется, признаки преобразуются в числовые значения, а входные значения нормализуются.
- *Обучение модели* — после сбора и предварительной обработки новых данных модели машинного обучения необходимо переобучить.
- *Развертывание модели* в рабочей среде может быть самой сложной частью перехода от машинного обучения, ориентированного на исследования, к продукту, основанному на реальных данных. Здесь необходимо не только управление версиями наборов данных, но и управление версиями обученных моделей. Часто для выполнения запросов к развернутой модели используют REST API и предусматривают разные конечные точки API для различных версий модели. За этим нужно внимательно следить, и чтобы перейти к стабильному рабочему состоянию, требуется координация между исследователями данных, инженерами по машинному обучению и инженерами данных. Хорошо спроектированный конвейер — ключ к успешному соединению составных частей.

Проверка собранных данных

Проверка данных в конвейере необходима и применяется во всех конвейерах. Об этом говорится в *главе 8*. В конвейерах, созданных для аналитиков, проверка часто происходит после сбора данных (извлечение-загрузка), а также после моделирования данных (преобразование). В конвейерах машинного обучения тоже важна проверка собранных данных. Не путайте этот важный шаг с проверкой самой модели ML, которая, конечно же, является стандартной частью разработки машинного обучения.

Включение обратной связи в конвейер

Любой хороший конвейер машинного обучения также предусматривает обратную связь для улучшения модели. Возьмем пример модели рекомендации контента для службы потокового видео. Чтобы улучшать модель в будущем, вам нужно отслеживать, что она рекомендует пользователям, какие рекомендации они выбирают и какой рекомендуемый контент им нравится после просмотра. Для этого вам нужно будет взаимодействовать с командой разработчиков, используя модель на главном экране потоковых сервисов. Им нужно будет реализовать какой-то тип сбора событий, который отслеживает каждую рекомендацию, сделанную каждому пользователю, установить версию модели, которая выдала рекомендацию, а также выяснить, какая рекомендация сработала. Затем следует перенести эту информацию в данные, связанные с потреблением контента пользователем, которые они наверняка уже собирают.

Далее всю эту информацию можно возвратить в хранилище данных и включить в будущие версии модели либо в качестве обучающих данных, либо для анализа и рассмотрения человеком (возможно, исследователем данных) для учета в будущей модели или эксперименте.

Кроме того, собранные данные могут быть обработаны, преобразованы и проанализированы аналитиками данных в соответствии с шаблоном ELT, описанным в этой книге. Аналитикам часто поручают измерить эффективность моделей и построить информационные панели для отображения ключевых показателей модели с точки зрения целей организации. Заинтересованные стороны могут использовать такие информационные панели, чтобы по-

нять, насколько эффективны различные модели для бизнеса и их клиентов.

Дополнительная литература по конвейерам машинного обучения

Построение конвейеров для моделей машинного обучения — серьезная тема. Есть несколько книг, которые я рекомендую для дальнейшего изучения в зависимости от вашего выбора инфраструктуры и сложности вашей среды машинного обучения:

- Ханнес Хапке, Кэтрин Нельсон. Разработка конвейеров машинного обучения. Автоматизация жизненных циклов модели с помощью TensorFlow. — М.: ДМК Пресс, 2021.
- Орельен Жерон. Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow. — М.: Вильямс, 2020.

Кроме того, следующая книга представляет собой очень доступное введение в машинное обучение:

- Андреас Мюллер, Сара Гвидо. Введение в машинное обучение с помощью Python. Руководство для специалистов по работе с данными. — М.: Вильямс, 2017.

ГЛАВА 4

Сбор данных: начнем с извлечения

Как обсуждалось в *главе 3*, шаблон ELT является идеальной основой для конвейеров данных, предназначенных для анализа, исследований и продуктов на основе данных. Первые два этапа шаблона ELT, извлечение и загрузка, в совокупности называются *сбором данных* (data ingestion). В этой главе рассматривается настройка среды разработки и инфраструктуры для обоих этапов, а также особенности извлечения данных из различных исходных систем. В *главе 5* обсуждается загрузка полученных наборов данных в хранилище.

> **ПРИМЕЧАНИЕ**
>
> Примеры кода извлечения и загрузки данных в этой главе полностью отделены друг от друга. Координация двух этапов для завершения сбора данных — это тема, которая обсуждается в *главе 7*.

Как упоминалось в *главе 2*, существует множество типов исходных систем, из которых можно извлечь данные, а также множество мест назначения, в которые их можно загрузить. Кроме того, данные поступают в разных формах, каждая из которых создает различные проблемы для их сбора.

Эта и следующая главы содержат примеры кода для экспорта и сбора данных из распространенных систем и отправки в них. Код сильно упрощен и содержит только минимальную обработку ошибок. Каждый пример задуман как простая для понимания отправная точка сбора данных, но он полностью функционален и расширяем до более масштабируемых решений.

> **ПРИМЕЧАНИЕ**
>
> Примеры кода в этой главе записывают извлеченные данные в файлы CSV для загрузки в целевое хранилище данных. Бывают

случаи, когда имеет смысл хранить извлеченные данные в другом формате, например JSON. Там, где это применимо, я отмечаю потенциальную возможность внесения такой корректировки.

В *главе 5* также рассматриваются некоторые фреймворки с открытым исходным кодом, на основе которых вы можете построить конвейер, и коммерческие альтернативы, которые предоставляют инженерам данных и аналитикам возможности реализации процесса сбора данных с помощью технологии *low-code* (с минимальным кодом).

Настройка среды Python

Все приведенные далее примеры кода написаны на Python и SQL и используют платформы с открытым исходным кодом, распространенные сегодня в области обработки данных. Для простоты число источников и мест назначения ограничено. Однако, где это применимо, я даю примечания о том, как можно модифицировать код.

Для запуска примеров кода вам понадобится физическая или виртуальная машина с Python 3.*x*. Вам также потребуется установить и импортировать несколько библиотек.

Если на вашем компьютере не установлен Python, вы можете получить дистрибутив и установщик для своей ОС непосредственно с сайта поддержки.

ПРИМЕЧАНИЕ

Следующие команды написаны для командной строки Linux или Macintosh. В Windows вам может потребоваться добавить путь к исполняемому файлу Python 3 в системную переменную `PATH`.

Перед установкой библиотек, используемых в этой главе, лучше всего создать *виртуальную среду* для их установки. Для этого подойдет инструмент под названием `virtualenv`, который помогает управлять библиотеками Python для различных проектов и приложений. Вы сможете устанавливать библиотеки Python в рамках определенного проекта, а не глобально. Сначала создайте виртуальную среду с именем `env`:

```
$ python -m venv env
```

Затем активируйте вновь созданную среду с помощью следующей команды:

```
$ source env/bin/activate
```

Существуют два способа убедиться, что ваша виртуальная среда активирована. Во-первых, вы заметите, что ваша командная строка теперь имеет префикс имени среды:

```
(env) $
```

Вы также можете выполнить команду `which python`, чтобы проверить, где Python ищет библиотеки. Вы должны увидеть что-то вроде следующего вывода в терминал, который показывает путь к каталогу виртуальной среды:

```
(env) $ which python
env/bin/python
```

Теперь можно безопасно установить библиотеки, необходимые для изучения следующих примеров кода.

> **ПРИМЕЧАНИЕ**
>
> В некоторых операционных системах (ОС) для запуска исполняемого файла Python 3.*x* необходима команда `python3` вместо `python`. Более старые версии ОС могут по умолчанию использовать Python 2.*x*. Вы можете узнать, какая версия Python задана в вашей ОС, набрав команду `python --version`.

В этой главе вам понадобится утилита `pip` — инструмент, который поставляется с большинством дистрибутивов Python для установки библиотек, используемых в примерах кода.

Первая библиотека, которую вы установите с помощью `pip`, — это configparser. Она пригодится для чтения информации о конфигурации, которую вы добавите в файл позже:

```
(env) $ pip установить configparser
```

Затем создайте файл с именем pipeline.conf в том же каталоге, что и сценарии Python, которые вы будете создавать в следующих разделах. Оставьте пока файл пустым. Примеры кода в этой главе постепенно наполнят его содержимым. В операционных системах Linux и Mac вы можете создать пустой файл в командной строке с помощью следующей команды:

```
(env) $ touch pipeline.conf
```

ВНИМАНИЕ!

Не добавляйте свои файлы конфигурации в репозиторий Git! Поскольку вы будете хранить учетные данные и информацию о подключении в файле конфигурации, не добавляйте его в свой репозиторий Git. Эта информация должна храниться только локально и в безопасных системах, которым разрешен доступ к корзине S3, исходным системам и хранилищу данных. Самый безопасный способ обеспечить исключение из репозитория — дать вашим файлам конфигурации специальное расширение, например .conf, и добавить в файл .gitignore строку с *.conf.

Настройка облачного хранилища файлов

В каждом примере в этой главе вы будете использовать для хранения файлов корзину Amazon Simple Storage Service (Amazon S3 или просто S3). S3 размещается на AWS, и, как следует из названия, — это простой способ хранения файлов и доступа к ним. Он также очень экономичен. На момент написания этой статьи AWS предлагает 5 ГБ бесплатного хранилища S3 на 12 месяцев с новой учетной записью AWS и взимает менее 3 центов США в месяц за гигабайт в стандартном хранилище класса S3 за последующий период. Учитывая простоту примеров в этой главе, вы сможете хранить необходимые данные в S3 бесплатно, если вы укладываетесь в первые 12 месяцев после создания учетной записи AWS, или менее чем за 1 доллар в месяц после этого.

Для запуска примеров в этой главе вам понадобится корзина S3. К счастью, создать корзину S3 несложно, а обновленные инструкции можно найти в документации AWS. Настройка надлежащего контроля доступа к корзине S3 зависит от вида вашего хранилища данных. Как правило, для политик управления доступом лучше всего подойдут роли AWS Identity and Access Management (IAM). Подробные инструкции по настройке такого доступа как для хранилища данных Amazon Redshift, так и для хранилища данных Snowflake приведены в следующих разделах, а пока следуйте инструкциям по созданию новой корзины. Назовите ее как угодно; я предлагаю оставить настройки по умолчанию, включая сохранение конфиденциальности корзины.

Каждый пример извлечения извлекает данные из заданной исходной системы и сохраняет выходные данные в корзине S3. Каждый пример загрузки в *главе 5* загружает эти данные из корзины S3 в место назначения. Это распространенный шаблон поведения в конвейерах данных. У каждого крупного поставщика общедоступных облачных сервисов есть сервис, аналогичный S3. Эквивалентами в других общедоступных облаках являются Azure Storage в Microsoft Azure и Google Cloud Storage (GCS) в GCP.

Каждый пример можно модифицировать для использования локального хранилища. Однако для загрузки данных в ваше хранилище из хранилища за пределами конкретного облачного провайдера требуется дополнительная работа. Несмотря на это, шаблоны, описанные в данной главе, действительны независимо от того, какой у вас облачный провайдер или инфраструктура размещения данных локально.

Прежде чем я перейду к конкретным примерам, вам нужно установить еще одну библиотеку Python, чтобы ваши скрипты для извлечения и загрузки могли взаимодействовать с вашей корзиной S3. Boto3 — это SDK AWS для Python. Убедитесь, что виртуальная среда, которую вы настроили в предыдущем разделе, активна, и используйте команду `pip` для установки библиотеки:

```
(env) $ pip install boto3
```

В следующих примерах вам будет предложено импортировать Boto3 в ваши скрипты Python следующим образом:

```
import boto3
```

Поскольку вам будет нужна библиотека Python Boto3 для взаимодействия с корзиной S3, также потребуется создать пользователя IAM, сгенерировать ключи доступа для этого пользователя и сохранить ключи в файле конфигурации, доступном для чтения из скриптов Python. Все это необходимо для того, чтобы ваши скрипты имели права на чтение и запись файлов в корзине S3.

Сначала создайте пользователя IAM:

1. В меню **Services** (Сервисы) в консоли AWS (или на верхней панели навигации) перейдите к IAM.
2. На панели навигации нажмите кнопку **Users** (Пользователи), а затем нажмите кнопку **Add user** (Добавить пользователя).

Введите имя нового пользователя. В этом примере имя пользователя data_pipeline_readwrite.

3. Нажмите кнопку **Type of access** (Тип доступа) для этого пользователя IAM. Нажмите кнопку **Programmatic access** (Программный доступ), т. к. этому пользователю не нужно будет входить в консоль AWS, а требуется получить программный доступ к ресурсам AWS с помощью скриптов Python.
4. Нажмите кнопку **Next: Permissions** (Далее: Разрешения).
5. На странице **Set permissions** (Установить разрешения) выберите параметр **Attach existing policies to user directly** (Прикрепить существующие политики непосредственно к пользователю). Добавьте политику **AmazonS3FullAccess**.
6. Нажмите кнопку **Next: Tags** (Далее: Теги). В AWS рекомендуется добавлять теги к различным объектам и службам, чтобы их можно было найти позже. Однако это необязательно.
7. Нажмите кнопку **Next: Review** (Далее: Просмотр), чтобы проверить настройки. Если все выглядит хорошо, нажмите **Create user** (Создать пользователя).
8. Вам нужно сохранить идентификатор ключа доступа и секретный ключ доступа для нового пользователя IAM. Для этого нажмите кнопку **Download .csv** (Загрузить .csv), а затем сохраните файл в безопасном месте, чтобы сразу же использовать его.

Наконец, добавьте в файл pipeline.conf раздел с именем `[aws_boto_credentials]` для хранения учетных данных пользователя IAM и информации о корзине S3. Вы можете найти идентификатор своей учетной записи AWS, нажав на имя своей учетной записи в правом верхнем углу любой страницы при входе на сайт AWS. Используйте имя корзины S3, которую вы создали ранее, в качестве значения параметра `bucket_name`. Новый раздел в pipeline.conf будет выглядеть так:

```
[aws_boto_credentials]
access_key = ijfiojr54rg8er8erg8erg8
secret_key = 5r4f84er4ghrg484eg84re84ger84
bucket_name = pipeline-bucket
account_id = 4515465518
```

Извлечение данных из БД MySQL

Извлечение данных из БД MySQL можно выполнить двумя способами:

- полное или инкрементное извлечение с использованием SQL;
- репликация двоичного журнала (Binary Log, binlog).

Первый способ гораздо проще реализовать, но он менее масштабируем для больших наборов данных с частыми изменениями. Существуют также компромиссы между полным и инкрементным извлечением, о которых я расскажу в следующем разделе.

Репликация двоичного журнала, хотя и сложнее для реализации, лучше подходит для случаев, когда объем изменяемых данных в исходных таблицах велик или существует потребность в более частом приеме данных из источника MySQL.

ПРИМЕЧАНИЕ

Репликация двоичного журнала также является путем к созданию приема потоковых данных. Дополнительную информацию о различиях между этими двумя подходами, а также о шаблонах реализации см. в *разделе "Пакетный и потоковый прием"* этой главы.

Этот раздел предназначен для читателей, располагающих источником данных MySQL, из которого нужно извлечь данные. Однако если вы хотите настроить простую базу данных, чтобы попробовать примеры кода, у вас есть два варианта. Во-первых, вы можете бесплатно установить MySQL на свой локальный или виртуальный компьютер. Вы найдете установщик для своей операционной системы на странице загрузок MySQL.

Кроме того, вы можете создать полностью управляемый экземпляр сервера Amazon RDS для MySQL в AWS. Я считаю этот метод более простым, и вдобавок приятно не создавать лишний беспорядок на своей локальной машине!

ПРЕДУПРЕЖДЕНИЕ

Когда вы будете следовать инструкциям по настройке экземпляра БД MySQL RDS, вам будет предложено сделать вашу базу данных общедоступной. Этого вполне достаточно для обучения и работы с демонстрационными данными. Фактически это значительно упрощает подключение с любой машины, на которой вы запускаете

> примеры из данного раздела. Однако для более надежной защиты в рабочей среде я предлагаю следовать рекомендациям по безопасности Amazon RDS.

Обратите внимание, что, как и в случае с тарифами на S3, упомянутыми ранее, если у вас истек бесплатный пробный период AWS, то использование RDS будет связано с затратами. В противном случае установка и запуск бесплатны! Просто не забудьте удалить свой экземпляр RDS, когда закончите, чтобы не забыть о нем и избежать расходов, когда закончится бесплатный пробный период.

Примеры кода в этом разделе довольно просты и ссылаются на таблицу `Orders` в БД MySQL. Когда у вас будет готов экземпляр MySQL для работы, вы можете создать таблицу и вставить несколько образцов строк, выполнив следующий SQL-запрос:

```
CREATE TABLE Orders (
  OrderId int,
  OrderStatus varchar(30),
LastUpdated timestamp
);

INSERT INTO Orders VALUES(1,'Размещен', '2020-06-01 12:00:00');
INSERT INTO Orders VALUES(1,'Отгружен', '2020-06-09 12:00:25');
INSERT INTO Orders VALUES(2,'Отгружен', '2020-07-11 3:05:00');
INSERT INTO Orders VALUES(1,'Отгружен', '2020-06-09 11:50:00');
INSERT INTO Orders VALUES(3,'Отгружен', '2020-07-12 12:00:00');
```

Полное или инкрементное извлечение таблицы MySQL

Если вам нужно загрузить все столбцы или подмножество столбцов из таблицы MySQL в хранилище или озеро данных, то сделать это можно, используя полное или инкрементное извлечение.

При *полном извлечении* каждый запуск задания извлечения приводит к извлечению всех записей из таблицы. Это наименее сложный подход, но при наличии больших таблиц его выполнение может занять много времени. Например, если вы хотите выполнить полное извлечение таблицы `Orders`, то SQL-запрос, выполненный в БД MySQL источника, будет выглядеть следующим образом:

```
SELECT *
FROM Orders;
```

При *инкрементном извлечении* из исходной таблицы извлекаются только те записи, которые были изменены или добавлены с момента последнего запуска задания. Метка времени последнего извлечения может быть либо сохранена в таблице журнала заданий извлечения в хранилище данных, либо получена путем запроса максимальной метки времени в столбце `LastUpdated` целевой таблицы в хранилище. Если продолжить пример с вымышленной таблицей `Orders`, то SQL-запрос, выполняемый в БД MySQL источника, будет выглядеть следующим образом:

```
SELECT *
FROM Orders
WHERE LastUpdated > {{ last_extraction_run} };
```

ПРИМЕЧАНИЕ

Для таблиц, содержащих неизменяемые данные (это означает, что записи могут быть добавлены, но не обновлены), вы можете использовать отметку времени создания записи вместо столбца `LastUpdated`.

Переменная `{{ last_extraction_run }}` представляет собой метку времени, отражающую самый последний запуск задания извлечения. Чаще всего она запрашивается из целевой таблицы в хранилище данных. В этом случае следующий запрос SQL будет выполняться в хранилище данных, а результирующее значение будет использоваться для обновления переменной `{{ last_extraction_run }}`:

```
SELECT MAX(LastUpdated)
FROM warehouse.Orders;
```

Кеширование дат последнего обновления

Если таблица `Orders` велика, вы можете сохранить значение последней обновленной записи в таблице журнала, которую можно будет быстро запросить при следующем запуске задания извлечения. Обязательно сохраните значение `MAX(LastUpdated)` из целевой таблицы в хранилище данных, а не время начала или завершения задания извлечения. Даже небольшая задержка во времени выполнения задания может означать пропущенные или дублированные записи из исходной таблицы при следующем запуске.

Хотя инкрементное извлечение идеально подходит для оптимизации производительности, существуют некоторые причины, по которым оно может быть неприменимо для данной таблицы.

Во-первых, при использовании этого метода не отслеживается удаление строк. Если строка будет удалена из исходной таблицы MySQL, вы не узнаете об этом, и она останется в целевой таблице, как будто ничего не изменилось.

Во-вторых, исходная таблица должна иметь надежную метку времени последнего обновления (столбец `LastUpdated` в предыдущем примере). В таблицах системы-источника такой столбец нередко отсутствует или не обновляется надежно. В результате может возникнуть ситуация, когда разработчики обновят записи в исходной таблице и забудут обновить метку времени `LastUpdated`.

Однако инкрементное извлечение упрощает отслеживание обновленных строк. В последующих примерах кода, если конкретная строка в таблице `Orders` будет обновлена, как полное, так и инкрементное извлечение вернут последнюю версию строки. При полном извлечении это справедливо для всех строк в таблице, поскольку извлекается полная копия таблицы. При инкрементном извлечении извлекаются только измененные строки.

Когда приходит время загрузки, полные извлечения обычно загружаются путем усечения целевой таблицы и загрузки вновь извлеченных данных. В этом случае у вас в хранилище данных останется только последняя версия строки.

При загрузке данных из инкрементного извлечения результирующие данные добавляются к данным в целевой таблице. В этом случае у вас есть как исходная запись, так и обновленная версия. Наличие и того и другого может оказаться полезным, когда придет время преобразовывать и анализировать данные, как будет показано в *главе 6*.

Например, в табл. 4.1 показана исходная запись для `OrderId 1` в БД MySQL. Когда клиент разместил заказ, он был в резерве. В табл. 4.2 показана обновленная запись в БД MySQL. Как видите, заказ был обновлен, т. к. он был отгружен 09.06.2020.

При выполнении полного извлечения целевая таблица в хранилище данных сначала очищается, а затем загружается выходными данными извлечения. Результатом для `OrderId 1` является единст-

венная запись, показанная в табл. 4.2. Однако при инкрементном извлечении выходные данные операции извлечения просто добавляются к целевой таблице в хранилище. В результате в хранилище данных находятся как исходные, так и обновленные записи для `OrderId 1`, как показано в табл. 4.3.

Таблица 4.1. *Исходное состояние OrderId 1*

OrderId	OrderStatus	LastUpdated
1	Размещен	2020-06-01 12:00:00

Таблица 4.2. *Обновленное состояние OrderId 1*

OrderId	OrderStatus	LastUpdated
1	Отгружен	2020-06-09 12:00:25

Таблица 4.3. *Все версии OrderId 1 в хранилище данных*

OrderId	OrderStatus	LastUpdated
1	Размещен	2020-06-01 12:00:00
1	Отгружен	2020-06-09 12:00:25

Вы можете узнать больше о загрузке полных и инкрементных извлечений в *главе 5*, включая *раздел "Загрузка данных в хранилище Redshift"*.

> **ПРЕДУПРЕЖДЕНИЕ**
>
> Никогда не надейтесь, что столбец `LastUpdated` в исходной системе надежно обновляется. Проконсультируйтесь с владельцем исходной системы и удостоверьтесь в этом, прежде чем использовать `LastUpdated` для инкрементного извлечения.

Как полное, так и инкрементное извлечение из БД MySQL может быть реализовано с помощью SQL-запросов, выполняемых в базе данных, но запускаемых сценариями Python. В дополнение к библиотекам Python, установленным в предыдущих разделах, вам необходимо установить библиотеку PyMySQL с помощью команды `pip`:

```
(env) $ pip install pymysql
```

Вам также потребуется добавить новый раздел в файл pipeline.conf для хранения информации о подключении к БД MySQL:

```
[mysql_config]
hostname = my_host.com
port = 3306
username = my_user_name
password = my_password
database = db_name
```

Теперь создайте новый скрипт Python с именем extract_mysql_full.py. Вам нужно будет импортировать несколько библиотек, таких как pymysql, которая подключается к БД MySQL, и библиотеку csv, чтобы вы могли структурировать и записывать извлеченные данные в плоский файл, который легко импортировать в хранилище данных на этапе выгрузки. Кроме того, импортируйте boto3, чтобы иметь возможность загрузить полученный CSV-файл в свою корзину S3 для последующей выгрузки в хранилище данных:

```
import pymysql
import csv
import boto3
import configparser
```

Теперь вы можете инициализировать подключение к БД MySQL:

```
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
hostname = parser.get("mysql_config", "hostname")
port = parser.get("mysql_config", "port")
username = parser.get("mysql_config", "username")
dbname = parser.get("mysql_config", "database")
password = parser.get("mysql_config", "password")

conn = pymysql.connect(host=hostname,
        user=username,
        password=password,
        db=dbname,
        port=int(port))

if conn is None:
   print("Ошибка подключения к базе данных MySQL")
```

```
else:
   print("Соединение с MySQL установлено!")
```

Запустите полное извлечение таблицы `Orders` из предыдущего примера. Следующий код извлечет все содержимое таблицы и запишет его в файл CSV с разделителями в виде вертикальной черты. Чтобы выполнить извлечение, он использует объект `cursor` из библиотеки pymysql для выполнения запроса SELECT:

```
m_query = "SELECT * FROM Orders;"
local_filename = "order_extract.csv"

m_cursor = conn.cursor()
m_cursor.execute(m_query)
results = m_cursor.fetchall()

with open(local_filename, 'w') as fp:
  csv_w = csv.writer(fp, delimiter='|')
  csv_w.writerows(results)

fp.close()
m_cursor.close()
conn.close()
```

Теперь, когда файл CSV записан локально, его необходимо поместить в корзину S3 для последующей выгрузки в хранилище данных или другое место назначения. Вспомните, что в *разделе "Настройка облачного хранилища файлов"* данной главы вы настроили пользователя IAM для библиотеки boto3, чтобы использовать его для аутентификации в корзине S3. Вы также сохранили учетные данные в разделе `aws_boto_credentials` файла pipeline.conf. Вот код для загрузки CSV-файла в корзину S3:

```
# Загрузка значений aws_boto_credentials
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
access_key = parser.get("aws_boto_credentials", "access_key")
secret_key = parser.get("aws_boto_credentials", "secret_key")
bucket_name = parser.get("aws_boto_credentials", "bucket_name")

s3 = boto3.client('s3',
aws_access_key_id=access_key,
aws_secret_access_key=secret_key)
```

```
s3_file = local_filename

s3.upload_file(local_filename, bucket_name, s3_file)
```

Вы можете выполнить скрипт следующим образом:

```
(env) $ python extract_mysql_full.py
```

После завершения работы скрипта все содержимое таблицы `Orders` теперь записано в файл CSV, который находится в корзине S3 и ожидает выгрузки в хранилище данных или другое место. В *главе 5* приводится дополнительная информация о загрузке в хранилище данных по вашему выбору.

Если вы хотите извлекать данные инкрементно, то нужно внести несколько изменений в скрипт. Я предлагаю создать копию скрипта extract_mysql_full.py с именем extract_mysql_incremental.py в качестве отправной точки.

Сначала найдите метку времени последней записи, извлеченной из исходной таблицы `Orders`. Для этого запросите значение `MAX(LastUpdated)` из таблицы `Orders` в хранилище данных. В этом примере я буду использовать хранилище данных Redshift (см. *раздел "Настройка хранилища Amazon Redshift в качестве места назначения"* в *главе 5*), но вы можете реализовать ту же логику с хранилищем по вашему выбору.

Для взаимодействия с кластером Redshift установите библиотеку psycopg2, если вы еще этого не сделали:

```
(env) $ pip install psycopg2
```

Далее представлен код для подключения и запроса кластера Redshift, чтобы получить значение `MAX(LastUpdated)` из таблицы `Orders`:

```
import psycopg2

# Получение информации о соединении с Redshift
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
dbname = parser.get("aws_creds", "database")
user = parser.get("aws_creds", "username")
password = parser.get("aws_creds", "password")
host = parser.get("aws_creds", "host")
port = parser.get("aws_creds", "port")
```

```
# Подключение к кластеру Redshift
rs_conn = psycopg2.connect(
    "dbname=" + dbname
    + " user=" + user
    + " password=" + password
    + " host=" + host
    + " port=" + port)

rs_sql = """SELECT COALESCE(MAX(LastUpdated),
        '1900-01-01')
        FROM Orders;"""
rs_cursor = rs_conn.cursor()
rs_cursor.execute(rs_sql)
result = rs_cursor.fetchone()

# Возвращаются только одна строка и один столбец
last_updated_warehouse = result[0]

rs_cursor.close()
rs_conn.commit()
```

Используя значение, хранящееся в `last_updated_warehouse`, измените выполнение запроса на извлечение в БД MySQL, чтобы извлекать из таблицы `Orders` только те записи, которые были обновлены с момента предыдущего выполнения задания на извлечение. Новый запрос содержит заполнитель `%s` для значения `last_updated_warehouse`. Затем значение передается в функцию `.execute()` библиотеки cursor в виде кортежа (тип данных, используемый для хранения коллекций данных). Это правильный и безопасный способ добавления параметров в запрос SQL, чтобы избежать возможной инъекции SQL. Так выглядит обновленный блок кода для выполнения SQL-запроса к БД MySQL:

```
m_query = """SELECT *
    FROM Orders
    WHERE LastUpdated > %s;"""
local_filename = "order_extract.csv"

m_cursor = conn.cursor()
m_cursor.execute(m_query, (last_updated_warehouse,))
```

Полный скрипт extract_mysql_incremental.py для инкрементного извлечения (с использованием кластера Redshift для значения last_updated) выглядит следующим образом:

```
import pymysql
import csv
import boto3
import configparser
import psycopg2

# Чтение данных аутентификации БД Redshift
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
dbname = parser.get("aws_creds", "database")
user = parser.get("aws_creds", "username")
password = parser.get("aws_creds", "password")
host = parser.get("aws_creds", "host")
port = parser.get("aws_creds", "port")

# Подключение к кластеру Redshift
rs_conn = psycopg2.connect(
    "dbname=" + dbname
    + " user=" + user
    + " password=" + password
    + " host=" + host
    + " port=" + port)

rs_sql = """SELECT COALESCE(MAX(LastUpdated),
        '1900-01-01')
        FROM Orders;"""
rs_cursor = rs_conn.cursor()
rs_cursor.execute(rs_sql)
result = rs_cursor.fetchone()

# Возвращаются только один столбец и одна строка
last_updated_warehouse = result[0]

rs_cursor.close()
rs_conn.commit()

# Аутентификация и подключение к MySQL
parser = configparser.ConfigParser()
```

```
parser.read("pipeline.conf")
hostname = parser.get("mysql_config", "hostname")
port = parser.get("mysql_config", "port")
username = parser.get("mysql_config", "username")
dbname = parser.get("mysql_config", "database")
password = parser.get("mysql_config", "password")

conn = pymysql.connect(host=hostname,
    user=username,
    password=password,
    db=dbname,
    port=int(port))

if conn is None:
  print("Ошибка подключения к БД MySQL")
else:
  print("Соединение с MySQL установлено!")

m_query = """SELECT *
      FROM Orders
      WHERE LastUpdated > %s;"""
local_filename = "order_extract.csv"

m_cursor = conn.cursor()
m_cursor.execute(m_query, (last_updated_warehouse,))
results = m_cursor.fetchall()

with open(local_filename, 'w') as fp:
  csv_w = csv.writer(fp, delimiter='|')
  csv_w.writerows(results)

fp.close()
m_cursor.close()
conn.close()

# Загрузка значений aws_boto_credentials
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
access_key = parser.get(
    "aws_boto_credentials",
    "access_key")
```

```
secret_key = parser.get(
    "aws_boto_credentials",
    "secret_key")
bucket_name = parser.get(
    "aws_boto_credentials",
    "bucket_name")
s3 = boto3.client(
    's3',
    aws_access_key_id=access_key,
    aws_secret_access_key=secret_key)

s3_file = local_filename

s3.upload_file(
    local_filename,
    bucket_name,
    s3_file)
```

ПРЕДУПРЕЖДЕНИЕ

Остерегайтесь больших заданий по извлечению (полных или инкрементных), которые создают нагрузку на исходную БД MySQL и даже блокируют выполнение производственных запросов. Проконсультируйтесь с владельцем БД и обсудите возможность настройки реплики для извлечения, чтобы не выполнять прямое извлечение из производственной БД.

Репликация двоичного журнала данных MySQL

Сбор данных из БД MySQL с использованием содержимого двоичного журнала MySQL для репликации изменений эффективен в случаях необходимости сбора больших объемов данных, хотя и более сложен в реализации.

ПРИМЕЧАНИЕ

Репликация двоичного журнала — это форма *отслеживания измененных данных* (Change Data Capture, CDC). Многие хранилища исходных данных поддерживают ту или иную форму CDC, которую вы можете использовать.

Двоичный журнал MySQL — это журнал, в котором хранятся записи о каждой операции, выполненной в БД. Например, в зависимости настройки, он может регистрировать все нюансы каждо-

го создания или изменения таблицы, а также каждой операции INSERT, UPDATE и DELETE. Хотя изначально он предназначался для репликации данных в другие экземпляры MySQL, нетрудно понять, почему содержимое двоичного журнала так привлекательно для инженеров данных, которые хотят собирать данные в свое хранилище.

Старайтесь воспользоваться готовой платформой

Из-за сложности репликации двоичных журналов я настоятельно рекомендую рассмотреть возможность использования платформы с открытым исходным кодом или коммерческого продукта, если вы хотите получать данные таким образом. Я рассматриваю один из таких вариантов в *разделе "Сбор потоковых данных с помощью Kafka и Debezium"* этой главы. Некоторые коммерческие инструменты, упомянутые далее в этой главе, также поддерживают работу с двоичными журналами.

Поскольку ваше хранилище данных, скорее всего, не является базой данных MySQL, встроенные функции репликации MySQL здесь неприменимы. Чтобы использовать двоичный журнал для приема данных в источник, отличный от MySQL, необходимо выполнить ряд шагов:

1. Включите и настройте журналирование на сервере MySQL.
2. Запустите начальное полное извлечение и загрузку таблицы.
3. Настройте извлечение из двоичного журнала на постоянной основе.
4. Обработайте и загрузите выдержки из двоичного журнала в хранилище данных.

ПРИМЕЧАНИЕ

Шаг 3 подробно не обсуждается, но чтобы реализовать сбор данных с помощью двоичного журнала, вы должны сначала заполнить таблицы в хранилище данных текущим состоянием БД MySQL, а затем использовать двоичный журнал для сбора последующих изменений. Для этого часто требуется установить параметр LOCK для таблиц, которые вы хотите извлечь, запустить mysqldump этих таблиц, а затем загрузить результат mysqldump в хранилище перед началом сбора данных при помощи двоичного журнала.

Хотя за инструкциями по включению и настройке двоичного журнала лучше всего обратиться к последней документации по MySQL, я бегло перечислю ключевые параметры конфигурации.

Проконсультируйтесь с владельцами системы-источника

Доступ к изменению конфигурации двоичных журналов в исходной системе MySQL часто предоставляется системным администраторам. Инженеры данных, которые хотят получить данные, должны обязательно проконсультироваться с владельцем БД, прежде чем пытаться изменить конфигурацию двоичного журнала, поскольку изменения могут повлиять на другие системы, а также на сам сервер MySQL.

Есть две ключевые настройки конфигурации двоичного журнала, которые необходимо обеспечить в БД MySQL.

Во-первых, убедитесь, что ведение двоичного журнала включено. Обычно оно включено по умолчанию, но вы можете проверить это, выполнив следующий SQL-запрос к БД (точный синтаксис может различаться в зависимости от дистрибутива MySQL):

```
SELECT variable_value as bin_log_status
FROM performance_schema.global_variables
WHERE variable_name='log_bin';
```

Если ведение журнала включено, вы увидите в окне терминала следующий текст:

```
+ - - - - - - - - - - - - - - - - - - -+
| bin_log_status :: |
+ - - - - - - - - - - - - - - - - - - -+
| ON |
+ - - - - - - - - - - - - - - - - - - -+
1 row in set (0.00 sec)
```

Если запрос возвращает статус `OFF`, вам необходимо обратиться к документации MySQL для соответствующей версии, чтобы включить журналирование.

Затем убедитесь, что установлен правильный формат ведения двоичного журнала. В последней версии MySQL поддерживаются три формата:

- `STATEMENT`;
- `ROW`;
- `MIXED`.

Формат `STATEMENT` регистрирует в журнале каждый оператор SQL, который вставляет или изменяет строку. Этот формат полезен,

если вы хотите реплицировать данные из одной БД MySQL в другую. Чтобы реплицировать данные, вы можете просто запустить все операторы и тем самым воспроизвести состояние БД. Однако, поскольку извлеченные данные, скорее всего, связаны с хранилищем данных, работающим на другой платформе, операторы SQL, созданные в БД MySQL, могут быть несовместимы с вашим хранилищем данных.

В формате `ROW` каждое изменение строки в таблице представляется в строке журнала не как оператор SQL, а как данные в самой строке. Это предпочтительный формат.

Формат `MIXED` вносит в журнал записи как в формате `STATEMENT`, так и `ROW`. Хотя позже можно отфильтровать только данные `ROW` (если двоичный журнал не служит для других целей), нет необходимости выбирать `MIXED`, потому что он занимает дополнительное дисковое пространство.

Вы можете узнать текущий формат журнала, выполнив следующий SQL-запрос:

```
SELECT variable_value as bin_log_format
FROM performance_schema.global_variables
WHERE variable_name='binlog_format';
```

Запрос выдаст формат, который в настоящее время активен:

```
+ - - - - - - - - - - - - - - - - - - - -+
| bin_log_format :: |
+ - - - - - - - - - - - - - - - - - - - -+
| ROW |
+ - - - - - - - - - - - - - - - - - - - -+
1 row in set (0.00 sec)
```

Формат двоичного журнала, а также другие параметры конфигурации обычно задаются в файле my.cnf, специфичном для экземпляра БД MySQL. Если вы откроете файл, то увидите следующую строку:

```
[mysqld]
binlog_format=row
........
```

Опять же, перед изменением каких-либо параметров конфигурации лучше проконсультироваться с владельцем БД MySQL или обратиться к последней версии документации MySQL.

Убедившись, что включено ведение двоичного журнала в формате ROW, вы можете создать процесс для извлечения из него соответствующей информации и сохранения ее в файле для загрузки в хранилище данных.

Из двоичного журнала в формате ROW вы будете извлекать события трех разных типов. В этом примере сбора данных вы можете игнорировать другие события, обнаруженные в журнале, но в более продвинутых стратегиях репликации также приходится извлекать события, которые изменяют структуру таблицы. Вы будете работать со следующими событиями:

- WRITE_ROWS_EVENT;
- UPDATE_ROWS_EVENT;
- DELETE_ROWS_EVENT.

Теперь пришло время получить события из журнала. К счастью, есть несколько библиотек Python с открытым исходным кодом, которые помогут вам начать работу. Одним из самых популярных является проект python-mysql-replication, который можно найти на GitHub. Для начала установите библиотеку с помощью команды pip:

```
(env) $ pip install mysql-replication
```

Чтобы получить представление о том, как выглядит вывод из двоичного журнала, вы можете подключиться к БД и выполнить чтение журнала. В приведенном далее примере (листинг 4.1) я буду использовать информацию о подключении к MySQL, добавленную в файл pipeline.conf для иллюстрации полного и инкрементного извлечения, рассмотренного ранее в этом разделе.

ПРИМЕЧАНИЕ

В листинге 4.1 выполняется чтение из файла двоичного журнала сервера MySQL, расположенного по умолчанию. Имя файла журнала и путь по умолчанию задаются в переменной log_bin, которая хранится в файле my.cnf БД MySQL. В некоторых случаях двоичные журналы меняются со временем (возможно, ежедневно или ежечасно). Если это так, вам нужно будет определить путь к файлу на основе метода чередования журналов и схемы именования файлов, выбранной администратором MySQL, и передать его в качестве значения параметра log_file при создании экземпляра

BinLogStreamReader. Дополнительные сведения приведены в документации по классу BinLogStream Reader.

Листинг 4.1. Пример работы с данными БД MySQL

```
from pymysqlreplication import BinLogStreamReader
from pymysqlreplication import row_event
import configparser
import pymysqlreplication

# Получение информации о подключении к MySQL
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
hostname = parser.get("mysql_config", "hostname")
port = parser.get("mysql_config", "port")
username = parser.get("mysql_config", "username")
password = parser.get("mysql_config", "password")
mysql_settings = {
"host": hostname,
"port": int(port),
"user": username,
"passwd": password
}
b_stream = BinLogStreamReader(
connection_settings = mysql_settings,
server_id=100,
only_events=[row_event.DeleteRowsEvent,
row_event.WriteRowsEvent,
row_event.UpdateRowsEvent]
)
for event in b_stream:
event.dump()
b_stream.close()
```

Есть несколько замечаний об объекте BinLogStreamReader, который создается в листинге 4.1. Сначала он подключается к БД MySQL, указанной в файле pipeline.conf, и выполняет чтение из заданного файла двоичного журнала. Далее, за счет комбинации параметра restore_stream=True и значения log_pos объект начинает чтение журнала с указанной точки. В данном случае это позиция 1400.

Наконец, я указываю `BinLogStreamReader` читать только события `DeleteRowsEvent`, `WriteRowsEvent` и `UpdateRowsEvent`, используя параметр `only_events`.

Затем скрипт перебирает все события и выводит их в удобочитаемом формате. Для вашей БД с таблицей `Orders` вы увидите приблизительно такой вывод:

```
=== WriteRowsEvent ===
Date: 2020-06-01 12:00:00
Log position: 1400
Event size: 30
Read bytes: 20
Table: orders
Affected columns: 3
Changed rows: 1
Values:
--
* OrderId : 1
* OrderStatus : Размещен
* LastUpdated : 2020-06-01 12:00:00

=== UpdateRowsEvent ===
Date: 2020-06-09 12:00:25
Log position: 1401
Event size: 56
Read bytes: 15
Table: orders
Affected columns: 3
Changed rows: 1
Affected columns: 3
Values:
--
* OrderId : 1 => 1
* OrderStatus : Размещен => Отгружен
* LastUpdated : 2020-06-01 12:00:00 => 2020-06-09
12:00:25
```

Как видите, есть два события, которые представляют `INSERT` и `UPDATE` для `OrderId 1`, как показано в табл. 4.3. В этом вымышленном примере два последовательных события журнала разделены днями, но на самом деле между ними будет множество событий, представляющих все изменения, внесенные в базу данных.

ПРИМЕЧАНИЕ

Значение `log_pos`, которое указывает объекту `BinLogStreamReader`, с какого места начать, вам нужно будет сохранить где-нибудь в собственной таблице, чтобы взять оттуда при следующем запуске извлечения. Я считаю, что лучше всего хранить значение в таблице журнала в хранилище данных, откуда оно может быть прочитано, когда начнется извлечение, и в которое оно может быть записано со значением позиции конечного события, когда оно завершится.

Хотя пример кода из листинга 4.1 показывает ход событий в удобочитаемом формате, чтобы упростить загрузку вывода в хранилище данных, необходимо сделать еще пару вещей:

- проанализировать и записать данные в другом формате. Чтобы упростить загрузку, в следующем примере кода каждое событие будет записано в строку CSV-файла;
- Записывайте по одному файлу для каждой таблицы, которую вы хотите извлечь и загрузить. Хотя пример журнала содержит только события, связанные с таблицей `Orders`, скорее всего в настоящий журнал будут включены события, связанные с другими таблицами.

Чтобы выполнить первое изменение, вместо вызова функции `.dump()` я буду извлекать атрибуты события и записывать их в файл формата CSV. Далее, вместо того, чтобы записывать файл для каждой таблицы, для простоты я буду записывать только события, связанные с таблицей `Orders`, в файл с именем orders_extract.csv. В окончательной версии кода извлечения данных необходимо сгруппировать события по таблицам и записать несколько файлов, по одному для каждой таблицы, для которой вы хотите получить изменения. На последнем этапе представленного далее примера кода (листинг 4.2) CSV-файл помещается в корзину S3, чтобы его можно было загрузить в хранилище данных, как подробно описано в *главе 5*.

Листинг 4.2. Модифицированный пример работы с данными БД MySQL

```
from pymysqlreplication import BinLogStreamReader
from pymysqlreplication import row_event
import configparser
```

```
import pymysqlreplication
import csv
import boto3

# Получение информации о подключении к MySQL
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
hostname = parser.get("mysql_config", "hostname")
port = parser.get("mysql_config", "port")
username = parser.get("mysql_config", "username")
password = parser.get("mysql_config", "password")

mysql_settings = {
    "host": hostname,
    "port": int(port),
    "user": username,
    "passwd": password
}

b_stream = BinLogStreamReader(
            connection_settings = mysql_settings,
            server_id=100,
            only_events=[row_event.DeleteRowsEvent,
                         row_event.WriteRowsEvent,
                         row_event.UpdateRowsEvent]
            )
order_events = []

for binlogevent in b_stream:
  for row in binlogevent.rows:
    if binlogevent.table == 'orders':
      event = {}
      if isinstance(
            binlogevent,row_event.DeleteRowsEvent
        ):
        event["action"] = "delete"
        event.update(row["values"].items())
      elif isinstance(
            binlogevent,row_event.UpdateRowsEvent
        ):
        event["action"] = "update"
        event.update(row["after_values"].items())
```

```
        elif isinstance(
              binlogevent,row_event.WriteRowsEvent
            ):
            event["action"] = "insert"
            event.update(row["values"].items())

        order_events.append(event)

b_stream.close()

keys = order_events[0].keys()
local_filename = 'orders_extract.csv'
with open(
        local_filename,
        'w',
        newline='') as output_file:
    dict_writer = csv.DictWriter(
                output_file, keys,delimiter='|')
    dict_writer.writerows(order_events)

# Чтение значений aws_boto_credentials
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
access_key = parser.get(
                "aws_boto_credentials",
                "access_key")
secret_key = parser.get(
                "aws_boto_credentials",
                "secret_key")
bucket_name = parser.get(
                "aws_boto_credentials",
                "bucket_name")

s3 = boto3.client(
    's3',
    aws_access_key_id=access_key,
    aws_secret_access_key=secret_key)

s3_file = local_filename

s3.upload_file(
    local_filename,
```

```
    bucket_name,
    s3_file)
```

После выполнения данного кода содержимое файла orders_extract.csv будет выглядеть так:

```
insert|1|Backordered|2020-06-01 12:00:00
update|1|Shipped|2020-06-09 12:00:25
```

Как мы узнаем далее из *главы 5*, формат результирующего CSV-файла оптимизирован для быстрой загрузки. Осмысление извлеченных данных — это задача этапа преобразования в конвейере, подробно рассмотренного в *главе 6*.

Извлечение данных из БД PostgreSQL

Как и в случае с MySQL, получить данные из БД PostgreSQL (известной также как Postgres) можно одним из двух способов: либо за счет полного или инкрементного извлечения с использованием SQL, либо путем вызова функций БД, предназначенных для поддержки репликации на другие узлы. При работе с Postgres есть несколько способов сделать это, но в данной главе основное внимание будет уделено одному из них: преобразованию *журнала упреждающих записей* (Write-Ahead Log, WAL) Postgres в поток данных.

Этот раздел в первую очередь адресован тем, кому необходимо получать данные из уже существующей БД Postgres. Однако если вы хотите просто попробовать примеры кода, то можете настроить Postgres либо путем установки на свой локальный компьютер, либо в AWS, используя экземпляр RDS, что я и рекомендую. В предыдущем разделе приведены примечания о тарифах и рекомендациях по обеспечению безопасности для RDS MySQL — эта информация применима и к RDS Postgres.

Примеры кода в этом разделе довольно просты и относятся к таблице `Orders` в БД Postgres. Получив настроенный экземпляр Postgres, вы можете создать таблицу и вставить несколько примеров строк, выполнив следующие команды SQL:

```
CREATE TABLE Orders (
  OrderId int,
```

```
  OrderStatus varchar(30),
  LastUpdated timestamp
);

INSERT INTO Orders VALUES(1,'Размещен', '2020-06-01 12:00:00');
INSERT INTO Orders VALUES(1,'Отгружен', '2020-06-09 12:00:25');
INSERT INTO Orders VALUES(2,'Отгружен', '2020-07-11 3:05:00');
INSERT INTO Orders VALUES(1,'Отгружен', '2020-06-09 11:50:00');
INSERT INTO Orders VALUES(3,'Отгружен', '2020-07-12 12:00:00');
```

Полное или инкрементное извлечение таблицы Postgres

Этот метод подобен инкрементному и полному извлечению, продемонстрированному в *разделе "Извлечение данных из БД MySQL"* этой главы. Он настолько похож, что я не буду вдаваться в подробности, кроме одного отличия в программном коде. Как и раньше, этот код будет извлекать данные из таблицы `Orders` в исходной БД, записывать их в файл CSV, а затем загружать в корзину S3.

Единственное отличие в этом разделе — это библиотека Python, которая потребуется для извлечения данных. Вместо PyMySQL я буду использовать для подключения к БД Postgres библиотеку pyscopg2. Если вы еще не установили ее, сделайте это с помощью команды `pip`:

```
(env) $ pip install pyscopg2
```

Вам также необходимо добавить новый раздел в файл pipeline.conf с информацией о подключении к БД Postgres:

```
[postgres_config]
host = myhost.com
port = 5432
username = my_username
password = my_password
database = db_name
```

Код для полного извлечения таблицы `Orders` (листинг 4.3) почти идентичен примеру из раздела про MySQL, но, как вы можете видеть, для подключения к исходной БД и выполнения запроса он использует библиотеку pyscopg2.

Листинг 4.3. Пример работы с данными БД Postgres

```
import psycopg2
import csv
import boto3
import configparser

parser = configparser.ConfigParser()
parser.read("pipeline.conf")
dbname = parser.get("postgres_config", "database")
user = parser.get("postgres_config", "username")
password = parser.get("postgres_config",
    "password")
host = parser.get("postgres_config", "host")
port = parser.get("postgres_config", "port")

conn = psycopg2.connect(
        "dbname=" + dbname
        + " user=" + user
        + " password=" + password
        + " host=" + host,
        port = port)

m_query = "SELECT * FROM Orders;"
local_filename = "order_extract.csv"

m_cursor = conn.cursor()
m_cursor.execute(m_query)
results = m_cursor.fetchall()

with open(local_filename, 'w') as fp:
  csv_w = csv.writer(fp, delimiter='|')
  csv_w.writerows(results)

fp.close()
m_cursor.close()
conn.close()

# Загрузка значений aws_boto_credentials
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
```

```
access_key = parser.get(
                "aws_boto_credentials",
                "access_key")
secret_key = parser.get(
                "aws_boto_credentials",
                "secret_key")
bucket_name = parser.get(
                "aws_boto_credentials",
                "bucket_name")
s3 = boto3.client(
      's3',
      aws_access_key_id = access_key,
      aws_secret_access_key = secret_key)

s3_file = local_filename

s3.upload_file(
    local_filename,
    bucket_name,
    s3_file)
```

Изменить инкрементную версию, показанную в разделе про MySQL, так же просто. Все, что вам нужно сделать, это использовать psycopg2 вместо PyMySQL.

Репликация данных с использованием журнала упреждающих записей

По аналогии с двоичным журналом MySQL, который обсуждался в предыдущем разделе, Postgres WAL можно использовать как метод CDC для извлечения. Как и в случае с двоичным журналом MySQL, сбор данных в конвейер с помощью WAL — довольно сложный процесс.

Вы можете применить упрощенный подход, аналогичный показанному ранее в качестве примера с двоичным журналом MySQL, но я предлагаю задействовать распределенную платформу с открытым исходным кодом под названием Debezium для потоковой передачи содержимого Postgres WAL в корзину S3 или в хранилище данных.

Хотя особенностям настройки и запуска служб Debezium стоит посвятить целую книгу, я даю краткий обзор Debezium и нюансы его использования для приема данных в конце главы в *разделе "Сбор потоковых данных с помощью Kafka и Debezium"*. Там вы можете узнать больше о том, как совместить работу с Debezium и Postgres CDC.

Извлечение данных из MongoDB

В следующем примере показано, как извлечь подмножество документов MongoDB из коллекции. В этом образце коллекции MongoDB документы представляют собой события, зарегистрированные в какой-либо системе, например веб-сервере. У каждого документа есть отметка времени его создания, а также ряд свойств, подмножество которых извлекается в рассмотренном далее примере кода. После завершения извлечения данные записываются в файл CSV и сохраняются в корзине S3, чтобы их можно было загрузить в хранилище данных на следующем этапе (см. *главу 5*).

Для подключения к БД MongoDB вам необходимо сначала установить библиотеку PyMongo. Как и в случае с другими библиотеками Python, вы можете установить ее с помощью команды `pip`:

```
(env) $ pip install pymongo
```

Конечно, вы можете изменить следующий пример кода, чтобы подключиться к вашему собственному экземпляру MongoDB и извлечь данные из ваших документов. Однако если вы хотите запустить код как есть, то можете сделать это, бесплатно создав кластер MongoDB с помощью MongoDB Atlas. Atlas — это полностью управляемая служба MongoDB, которая включает в себя бесплатный уровень с большим объемом памяти и вычислительной мощностью для обучения и запуска примеров, подобных тому, который я предоставляю. Для развертывания производственных приложений вы можете в любой момент перейти на платный уровень.

Следуя инструкциям на сайте, вы узнаете, как создать бесплатный кластер MongoDB в Atlas, создать БД и настроить ее так, чтобы вы могли подключаться через скрипт Python, работающий на вашем локальном компьютере.

Вам нужно будет установить еще одну библиотеку Python с именем dnspython для поддержки pymongo при подключении к вашему кластеру, размещенному в MongoDB Atlas. Сделайте это с помощью `pip`:

```
(env) $ pip install dnspython
```

Затем добавьте новый раздел в файл pipeline.conf с информацией о подключении к экземпляру MongoDB, из которого вы будете извлекать данные:

```
[mongo_config]
hostname = my_host.com
username = mongo_user
password = mongo_password
database = my_database
collection = my_collection
```

Заполните каждую строку своими данными подключения. Если вы используете MongoDB Atlas и не можете вспомнить значения, которые вводили при настройке кластера, то узнать, как их найти, можно, прочитав документацию Atlas.

Перед созданием и запуском скрипта извлечения вы можете вставить несколько образцов данных для работы. Создайте файл с именем sample_mongodb.py со следующим содержимым:

```
from pymongo import MongoClient
import datetime
import configparser

# Чтение значений mongo_config
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
hostname = parser.get("mongo_config", "hostname")
username = parser.get("mongo_config", "username")
password = parser.get("mongo_config", "password")
database_name = parser.get("mongo_config", "database")
collection_name = parser.get("mongo_config", "collection")
mongo_client = MongoClient(
                "mongodb+srv://" + username
                + ":" + password
                + "@" + hostname
                + "/" + database_name
```

```
                + "?retryWrites=true&"
                + "w=majority&ssl=true&"
                + "ssl_cert_reqs=CERT_NONE")

# Подключение к БД, в которой хранится коллекция
mongo_db = mongo_client[database_name]

# Выбор коллекции для извлечения документов
mongo_collection = mongo_db[collection_name]

event_1 = {
  "event_id": 1,
  "event_timestamp": datetime.datetime.today(),
  "event_name": "signup"
}

event_2 = {
  "event_id": 2,
  "event_timestamp": datetime.datetime.today(),
  "event_name": "pageview"
}

event_3 = {
  "event_id": 3,
  "event_timestamp": datetime.datetime.today(),
  "event_name": "login"
}

# Вставка трех документов
mongo_collection.insert_one(event_1)
mongo_collection.insert_one(event_2)
mongo_collection.insert_one(event_3)
```

После запуска скрипта в вашу коллекцию MongoDB будут вставлены три документа:

```
(env) $ python sample_mongodb.py
```

Теперь создайте новый скрипт Python с именем mongo_extract.py, чтобы добавить в него следующие блоки кода.

Сначала импортируйте PyMongo и Boto3, чтобы можно было извлекать данные из БД MongoDB и сохранять результаты в корзине S3. Также импортируйте библиотеку csv, чтобы можно было

структурировать и записывать извлеченные данные в плоский файл, который легко импортировать в хранилище на этапе загрузки сбора данных. Наконец, для этого примера вам потребуются некоторые функции даты и времени, чтобы вы могли перебирать данные событий в коллекции MongoDB:

```
from pymongo import MongoClient
import csv
import boto3
import datetime
from datetime import timedelta
import configparser
```

Затем подключитесь к экземпляру MongoDB, указанному в файле pipeline.conf, и создайте объект `collection` для хранения извлекаемых документов:

```
# Чтение данных mongo_config
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
hostname = parser.get("mongo_config", "hostname")
username = parser.get("mongo_config", "username")
password = parser.get("mongo_config", "password")
database_name = parser.get("mongo_config", "database")
collection_name = parser.get("mongo_config", "collection")

mongo_client = MongoClient(
                "mongodb+srv://" + username
                + ":" + password
                + "@" + hostname
                + "/" + database_name
                + "?retryWrites=true&"
                + "w=majority&ssl=true&"
                + "ssl_cert_reqs=CERT_NONE")

# Подключение к БД, в которой хранится коллекция
mongo_db = mongo_client[database_name]

# Выбор коллекции для извлечения документов
mongo_collection = mongo_db[collection_name]
```

Теперь пришло время запросить документы для извлечения. Вы можете сделать это, вызвав функцию `.find()` в `mongo_collection`,

чтобы запросить документы, которые вы ищете. В следующем примере вы получите все документы со значением поля event_timestamp между двумя датами, определенными в скрипте:

```
start_date = datetime.datetime.today() + timedelta(days = -1)
end_date = start_date + timedelta(days = 1 )

mongo_query = { "$and":[{"event_timestamp" :
{ "$gte": start_date }}, {"event_timestamp" :
{ "$lt": end_date }}] }
event_docs = mongo_collection.find(mongo_query, batch_size=3000)
```

ПРИМЕЧАНИЕ

Извлечение неизменяемых данных, таких как записи журнала или общие записи "событий" из хранилища данных по диапазону дат, является распространенным вариантом использования. Хотя в показанном здесь примере диапазон дат и времени жестко задан в коде скрипта, более вероятно, что вы передадите диапазон дат и времени скрипту или попросите скрипт запросить ваше хранилище данных, чтобы получить дату и время последнего загруженного события и извлечь более новые записи из хранилища исходных данных. См. *раздел "Извлечение данных из БД MySQL"* этой главы в качестве примера.

ПРИМЕЧАНИЕ

Для параметра batch_size в этом примере установлено значение 3000. PyMongo выполняет двусторонний обмен данными с хостом MongoDB для каждого пакета. Например, если буфер result_docs Cursor содержит 6000 результатов, потребуется два обращения к хосту MongoDB, чтобы перенести все документы на компьютер, на котором выполняется ваш скрипт Python. Размер пакета определяется вами и зависит от компромисса между хранением большего количества документов в памяти в системе, выполняющей извлечение, и выполнением большого числа циклов обращения к экземпляру MongoDB.

Результат выполнения предыдущего фрагмента кода — объект Cursor с именем event_docs, который я буду использовать для перебора результирующих документов. Напомню, что в этом упрощенном примере каждый документ представляет собой событие, созданное такой системой, как веб-сервер. Событие может представлять собой что-то вроде входа пользователя в систему, просмотра страницы или отправки формы обратной связи. Хотя

в документах могут быть десятки полей для представления такой информации, как браузер, с помощью которого пользователь вошел в систему, я возьму для этого примера всего несколько полей:

```
# Создаем пустой список для сохранения результатов
all_events = []

# Перебираем cursor
for doc in event_docs:
    # Включаем значения по умолчанию
    event_id = str(doc.get("event_id", -1))
    event_timestamp = doc.get("event_timestamp", None)
    event_name = doc.get("event_name", None)

    # Добавляем все свойства события в список
    current_event = []
    current_event.append(event_id)
    current_event.append(event_timestamp)
    current_event.append(event_name)

    # Добавляем событие в окончательный список событий
    all_events.append(current_event)
```

Я включил значение по умолчанию в вызов функции `doc.get()` (`-1` или `None`). Почему? Само понятие неструктурированных данных документа означает, что поля могут вообще отсутствовать в документе. Другими словами, вы не можете рассчитывать, что каждый из документов, которые вы просматриваете, имеет `event_name` или любое другое поле. В таких случаях `doc.get()` вернет значение `None` вместо выдачи сообщения об ошибке.

После перебора всех событий в `event_docs` список `all_events` готов для записи в CSV-файл. Для этого мы будем использовать модуль `csv`, который входит в стандартный дистрибутив Python и был импортирован ранее в этом примере:

```
export_file = "export_file.csv"

with open(export_file, 'w') as fp:
        csvw = csv.writer(fp, delimiter='|')
        csvw.writerows(all_events)

fp.close()
```

Теперь загрузите CSV-файл в корзину S3, которую вы настроили в *разделе "Настройка облачного хранилища файлов"* данной главы. Для этого используйте библиотеку Boto3:

```
# Чтение значений aws_boto_credentials
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
access_key = parser.get("aws_boto_credentials", "access_key")
secret_key = parser.get("aws_boto_credentials", "secret_key")
bucket_name = parser.get("aws_boto_credentials", "bucket_name")

s3 = boto3.client('s3',
        aws_access_key_id=access_key,
        aws_secret_access_key=secret_key)

s3_file = export_file

s3.upload_file(export_file, bucket_name, s3_file)
```

Вот и все! Данные, которые вы извлекли из коллекции MongoDB, теперь находятся в корзине S3, ожидая загрузки в хранилище данных. Если вы взяли предоставленные образцы данных, то содержимое файла export_file.csv будет выглядеть так:

```
1|2020-12-13 11:01:37.942000|signup
2|2020-12-13 11:01:37.942000|pageview
3|2020-12-13 11:01:37.942000|login
```

В *главе 5* будет рассказано о загрузке данных в выбранное вами хранилище.

Извлечение данных из REST API

REST API — распространенный источник для извлечения данных. Вам может потребоваться получить данные из API, созданного и поддерживаемого вашей организацией, или из API внешней службы или поставщика, с которым взаимодействует ваша организация, например Salesforce, HubSpot или Twitter. Независимо от API, есть общий шаблон извлечения данных, который я буду использовать в следующем простом примере:

1. Отправьте запрос HTTP GET на конечную точку API.
2. Получите ответ, скорее всего, в формате JSON.

3. Проанализируйте ответ и "сведите" его в файл CSV, который впоследствии можно будет загрузить в хранилище данных.

> **ПРИМЕЧАНИЕ**
>
> Хотя я анализирую ответ JSON и сохраняю его в неструктурированном файле (CSV), вы также можете сохранить данные в формате JSON для последующей загрузки в свое хранилище данных. Для простоты я следую традициям этой главы и использую файлы CSV. Дополнительные сведения о загрузке данных в формате, отличном от CSV, приведены в *главе 5* или в документации по вашему хранилищу данных.

В этом примере я буду подключаться к API под названием Open Notify. У API есть несколько конечных точек, каждая из которых возвращает данные от НАСА о том, что происходит в космосе. Я запрошу конечную точку, которая возвращает следующие пять значений времени, когда Международная космическая станция (МКС) пролетит над заданным местом на Земле.

Библиотеки Python для определенных API

Используя пример кода Python в этом разделе, можно запросить любой REST API. Однако вы можете сэкономить время и силы, если есть библиотека Python, созданная специально для API, который вы хотите запрашивать. Например, библиотека tweepy упрощает для разработчика Python доступ к Twitter API и работу с распространенными структурами данных Twitter, такими как твиты и пользователи.

Прежде чем я поделюсь кодом Python для запроса конечной точки, предлагаю взглянуть на результат простого запроса, введя следующий URL-адрес в свой браузер:

```
http://api.open-notify.org/iss-pass.json?
lat=42.36&lon=71.05
```

Полученный JSON выглядит так:

```
{
  "message": "success",
  "request": {
  "altitude": 100,
  "datetime": 1596384217,
  "latitude": 42.36,
  "longitude": 71.05,
  "passes": 5
},
```

```
"response": [
  {
    "duration": 623,
    "risetime": 1596384449
  },
  {
    "duration": 169,
    "risetime": 1596390428
  },
  {
    "duration": 482,
    "risetime": 1596438949
  },
  {
    "duration": 652,
    "risetime": 1596444637
  },
  {
    "duration": 624,
    "risetime": 1596450474
  }
 ]
}
```

Цель этого запроса — извлечение данных из ответа и их запись в CSV-файл в виде одной строки для каждого времени и продолжительности каждого прохода, который ISS будет выполнять над точкой с заданной широтой и долготой. Например, первые две строки CSV-файла будут такими:

```
42.36,|71.05|623|1596384449
42.36,|71.05|169|1596390428
```

Чтобы запросить API и обработать ответ в Python, вам необходимо установить библиотеку requests. Она упрощает работу с HTTP-запросами и ответами в Python. Установите ее с помощью `pip`:

```
(env) $ pip install requests
```

Теперь с помощью requests вы можете запросить конечную точку API, получить ответ и распечатать полученный JSON, который будет выглядеть так, как вы видели в своем браузере:

```
import requests

lat = 42.36
lon = 71.05
lat_log_params = {"lat": lat, "lon": lon}

api_response = requests.get(
"http://api.open-notify.org/iss-pass.json", params=lat_log_params)

print(api_response.content)
```

Вместо того, чтобы распечатывать JSON, я перебираю поля ответа, анализирую значения длительности и времени восхода, записываю результаты в CSV-файл и загружаю его в корзину S3.

Чтобы выполнить синтаксический разбор ответа JSON, я импортирую библиотеку Python json. Она не нуждается в установке, т. к. поставляется в стандартном дистрибутиве Python. Далее я импортирую библиотеку csv, которая также входит в стандартный дистрибутив Python, для записи CSV-файла. Наконец, я буду использовать библиотеку configparser, чтобы получить учетные данные, необходимые библиотеке Boto3 для загрузки CSV-файла в корзину S3:

```
import requests
import json
import configparser
import csv
import boto3
```

Затем я запрашиваю API так же, как и раньше:

```
lat = 42.36
lon = 71.05
lat_log_params = {"lat": lat, "lon": lon}

api_response = requests.get(
    "http://api.open-notify.org/iss-pass.json",
                        params=lat_log_params)
```

Теперь можно выполнить синтаксический разбор ответа, поместить результаты в список Python с именем `all_passes` и сохранить результаты в файл формата CSV. Обратите внимание, что я также сохраняю широту и долготу из запроса, даже если они не вклю-

чены в ответ. Они нужны в каждой строке CSV-файла, чтобы моменты времени прохождения МКС были связаны с правильными широтой и долготой при загрузке в хранилище данных:

```
# Создание объекта json из содержимого ответа
response_json = json.loads(api_response.content)

all_passes = []
for response in response_json['response']:
    current_pass = []

    # Сохранение широты и долготы из запроса
    current_pass.append(lat)
    current_pass.append(lon)

    # Сохранение длительности и времени восхода для точки
    current_pass.append(response['duration'])
    current_pass.append(response['risetime'])

    all_passes.append(current_pass)

export_file = "export_file.csv"

with open(export_file, 'w') as fp:
        csvw = csv.writer(fp, delimiter='|')
        csvw.writerows(all_passes)

fp.close()
```

Наконец, загрузим CSV-файл в корзину S3, применив библиотеку Boto3:

```
# Загрузка значений aws_boto_credentials
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
access_key = parser.get("aws_boto_credentials", "access_key")
secret_key = parser.get("aws_boto_credentials", "secret_key")
bucket_name = parser.get("aws_boto_credentials", "bucket_name")

s3 = boto3.client(
    's3',
    aws_access_key_id=access_key,
    aws_secret_access_key=secret_key)
```

```
s3.upload_file(
    export_file,
    bucket_name,
    export_file)
```

Сбор потоковых данных с помощью Kafka и Debezium

Когда дело доходит до приема данных из системы CDC, такой как двоичные журналы MySQL или WAL Postgres, не существует простого решения, работающего без поддержки мощной инфраструктуры.

Debezium — это распределенная система, состоящая из нескольких служб с открытым исходным кодом, которые фиксируют изменения на уровне строк из обычных систем CDC, а затем передают их в виде событий, которые могут быть доступны другим системам. Платформа Debezium состоит из трех основных компонентов:

- *Apache Zookeeper* управляет распределенной средой и конфигурирует каждую службу.
- *Apache Kafka* — это распределенная платформа потоковой передачи данных, которая обычно применяется для создания масштабируемых конвейеров данных.
- *Apache Kafka Connect* — это инструмент-коннектор для подключения Kafka к другим системам, чтобы можно было легко передавать данные через Kafka. *Коннекторы* создаются для таких систем, как MySQL и Postgres, и превращают данные из их систем CDC (двоичные журналы и WAL) в *топики Kafka*.

Kafka обменивается сообщениями, организованными по *топикам* (topic). Одна система может публиковать сообщения в топике, в то время как другие могут подписываться на топик и читать сообщения.

Debezium связывает эти системы вместе и содержит коннекторы для общих реализаций CDC. Например, мы обсуждали проблемы работы с CDC в *разделах "Извлечение данных из БД MySQL"* и *"Извлечение данных из БД PostgreSQL"* этой главы. К счастью,

уже созданы коннекторы для "прослушивания" бинарного журнала MySQL и Postgres WAL. Затем данные направляются через Kafka в виде публикаций в топике и передаются в место назначения, например в корзину S3, хранилище данных Snowflake или Redshift, с помощью другого коннектора. На рис. 4.1 проиллюстрирован пример использования Debezium и его отдельных компонентов для отправки событий, созданных двоичным журналом MySQL, в хранилище данных Snowflake.

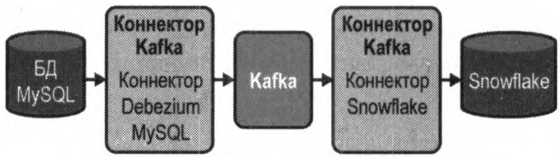

Рис. 4.1. Использование компонентов Debezium для CDC из MySQL в Snowflake

На момент написания этой книги уже было создано несколько коннекторов Debezium для исходных систем, которые вам могут понадобиться:

- MongoDB;
- MySQL;
- PostgreSQL;
- Microsoft SQL Server;
- Oracle;
- Db2;
- Cassandra.

Существуют также коннекторы Kafka Connect для наиболее распространенных хранилищ данных и систем хранения, таких как S3 и Snowflake.

Хотя Debezium и сама Kafka — это темы, которые заслуживают отдельной книги, я хочу указать на их ценность, если вы решите использовать для приема данных метод CDC. Простой пример, который я привел в разделе про извлечение данных из MySQL, вполне функционален. Но если вы хотите реализовать CDC на

уровне производства, я настоятельно рекомендую выбрать что-то вроде Debezium, а не создавать существующую платформу, такую как Debezium, самостоятельно!

СОВЕТ

Документация Debezium превосходна и является отличной отправной точкой для изучения системы.

Сбор данных: загрузка в хранилище

В *главе 4* вы узнали, как извлечь данные из нужной исходной системы. Теперь пришло время завершить сбор данных, загрузив их в хранилище данных Redshift. Способ загрузки зависит от того, как выглядит результат извлечения данных. В этом разделе будет показано, как загружать данные, извлеченные в CSV-файлы со значениями, соответствующими каждому столбцу в таблице, а также данные в формате CDC.

Настройка хранилища Amazon Redshift в качестве места назначения

Если для хранения своих данных вы используете Amazon Redshift, интеграция с S3 для загрузки данных после их извлечения довольно проста. Первый шаг — создать роль IAM для загрузки данных, если у вас ее еще нет.

> **ПРИМЕЧАНИЕ**
>
> Инструкции по настройке кластера Amazon Redshift вы можете найти в фирменной документации. Там же есть информация о тарифах, включая бесплатные пробные планы.

> **Не путайте роли IAM и пользователей IAM**
>
> В *разделе "Настройка облачного хранилища файлов" главы 4* вы создали *пользователя* IAM с правами на чтение и запись в корзине S3, которую вы будете использовать в этом разделе. В этом разделе вы создадите *роль* IAM, которой назначите разрешения, специфичные для чтения из S3 непосредственно в ваш кластер Redshift.

Чтобы создать роль, следуйте приведенным далее инструкциям или ознакомьтесь с последней информацией в документации AWS:

1. В меню **Services** (Сервисы) консоли AWS (или на верхней панели навигации) перейдите к разделу **IAM**.
2. В левом навигационном меню выберите пункт **Roles** (Роли) и нажмите кнопку **Create role** (Создать роль).
3. Вам будет представлен список сервисов AWS на выбор. Найдите и выберите Redshift.
4. В поле **Select your use case** (Выберите вариант использования) выберите опцию **Redshift — Customizable** (Redshift — Настраиваемый).
5. На следующей странице **Attach permission policies** (Связать политики разрешений) найдите и выберите опцию **AmazonS3ReadOnlyAccess** и нажмите кнопку **Next** (Далее).
6. Дайте вашей роли имя (например, RedshiftLoadRole) и нажмите кнопку **Create role** (Создать роль).
7. Нажмите на имя новой роли и скопируйте имя ресурса Amazon (Amazon Resource Name, ARN), чтобы использовать его далее в этой главе. Вы также можете найти его позже в консоли IAM в свойствах роли. ARN выглядит так:

```
arn:aws:iam::<aws-account-id>:role/<role-name>
```

Теперь вы можете связать только что созданную роль IAM со своим кластером Redshift. Для этого выполните приведенные далее действия или ознакомьтесь с документацией Redshift для получения более подробной информации.

> **ПРИМЕЧАНИЕ**
>
> Кластеру потребуется минута или две, чтобы применить изменения, но он будет доступен в течение этого времени.

1. Вернитесь в меню **Services** панели AWS и перейдите в раздел **Amazon Redshift**.
2. В меню навигации выберите пункт **Clusters** (Кластеры) и выберите кластер, в который вы хотите загрузить данные.
3. В разделе **Actions** (Действия) нажмите кнопку **Manage IAM roles** (Управление ролями IAM).
4. На загружаемой странице **Manage IAM roles** вы сможете выбрать свою роль в раскрывающемся списке **Available roles**

(Доступные роли). Затем нажмите кнопку **Add IAM role** (Добавить роль IAM).

5. Нажмите кнопку **Done** (Готово).

Наконец, добавьте еще один раздел в файл pipeline.conf, который вы создали в разделе *главы 4 "Настройка облачного хранилища файлов"*, с вашими учетными данными Redshift и именем только что созданной роли IAM. Информацию о подключении к кластеру Redshift можно найти на странице консоли AWS Redshift:

```
[aws_creds]
database = my_warehouse
username = pipeline_user
password = weifj4tji4j
host = my_example.4754875843.useast-1.redshift.amazonaws.com
port = 5439
iam_role = RedshiftLoadRole
```

Рекомендации по безопасности учетных данных Redshift

Для простоты в этом примере для подключения к кластеру из кода Python используются имя пользователя и пароль базы данных. В производственной среде следует предусмотреть более надежные стратегии безопасности, включая аутентификацию IAM для создания временных учетных данных БД. Вы можете узнать больше по адресу **https://oreil.ly/VwQHX**. Вместо локального файла pipeline.conf вы также можете воспользоваться возможностью более безопасного хранения учетных данных БД и других секретов. Один из популярных вариантов — сервис Vault.

Загрузка данных в хранилище Redshift

Загрузка в Redshift данных в виде CSV-файла, которые были извлечены и сохранены в виде значений, соответствующих каждому столбцу таблицы в корзине S3, относительно проста. Данные в этом формате наиболее распространены и являются результатом извлечения из такого источника, как БД MySQL или MongoDB. Каждая строка в загружаемом CSV-файле соответствует записи, которую нужно загрузить в целевую таблицу Redshift, а каждый столбец в формате CSV соответствует столбцу в целевой таблице. Если вы извлекли события из двоичного журнала MySQL или

другого журнала CDC, то обратитесь к следующему разделу за инструкциями по загрузке.

Самый эффективный способ загрузить данные из S3 в Redshift — применить команду `COPY`. Эта команда может выполняться как оператор SQL в любом клиенте SQL, который вы выбрали для запросов к кластеру Redshift, или в сценарии Python с использованием библиотеки Boto3. Команда `COPY` добавляет данные, которые вы загружаете, к существующим строкам в целевой таблице.

Использование редактора запросов Redshift

Самый простой способ сделать запрос к кластеру Redshift — использовать редактор запросов, встроенный в консольное веб-приложение AWS. Хотя его возможности ограничены, вы можете редактировать, сохранять и выполнять SQL-запросы к вашему кластеру прямо в браузере. Чтобы получить к нему доступ, войдите в консоль Redshift и нажмите кнопку **EDITOR** (РЕДАКТОР) в окне навигации.

Синтаксис команды `COPY` следующий (все элементы в квадратных скобках `[]` являются необязательными):

```
COPY table_name
[ column_list ]
FROM source_file
authorization
[ [ FORMAT ] [ AS ] data_format ]
[ parameter [ argument ] [, .. ] ]
```

ПРИМЕЧАНИЕ

Вы можете узнать больше о дополнительных параметрах и команде `COPY` в целом в документации AWS.

Простейшая форма авторизации роли IAM, как описано в *главе 4*, и файла в вашей корзине S3 при запуске из клиента SQL выглядит примерно так:

```
COPY my_schema.my_table
FROM 's3://bucket-name/file.csv'
iam_role '<my-arn>';
```

Как вы помните из *раздела "Настройка хранилища Amazon Redshift в качестве места назначения"* этой главы, ARN имеет следующий формат:

```
arn:aws:iam::<aws-account-id>:role/<role-name>
```

Если вы назвали роль `RedshiftLoadRole`, то синтаксис команды `COPY` выглядит следующим образом:

```
COPY my_schema.my_table
FROM 's3://bucket-name/file.csv'
iam_role 'arn:aws:iam::222:role/RedshiftLoadRole';
```

Обратите внимание, что числовое значение в ARN относится к вашей учетной записи AWS.

При выполнении команды содержимое файла file.csv добавляется в таблицу с именем `my_table` в схеме `my_schema` вашего кластера Redshift.

По умолчанию команда `COPY` вставляет данные в столбцы целевой таблицы в том же порядке, что и поля во входном файле. Другими словами, если вы не укажете иное, порядок полей в CSV-файле, который вы загружаете в этом примере, должен соответствовать порядку столбцов в целевой таблице в Redshift. Если вы хотите указать порядок столбцов, то это можно сделать, добавив имена столбцов назначения в порядке, соответствующем вашему входному файлу:

```
COPY my_schema.my_table (column_1, column_2, ....)
FROM 's3://bucket-name/file.csv'
iam_role 'arn:aws:iam::222:role/RedshiftLoadRole';
```

Также для реализации команды `COPY` в скрипте Python можно использовать библиотеку Boto3. На самом деле, если следовать шаблону примеров извлечения данных, рассмотренному в *главе 4*, загрузка данных через Python делает конвейер данных более стандартизированным.

Для взаимодействия с кластером Redshift, который вы настроили ранее в этой главе, вам потребуется установить библиотеку psycopg2:

```
(env) $ pip install psycopg2
```

Теперь вы можете начать писать свой скрипт Python. Создайте новый файл с именем copy_to_redshift.py и добавьте следующие три блока кода.

Первый шаг — импорт `boto3` для взаимодействия с корзиной S3, `psycopg2` для запуска команды `COPY` в кластере Redshift и библиотеки `configparser` для чтения файла pipeline.conf:

```
import boto3
import configparser
import psycopg2
```

Затем подключитесь к кластеру Redshift, используя функцию `psycopg2.connect` и учетные данные, хранящиеся в файле pipeline.conf:

```
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
dbname = parser.get("aws_creds", "database")
user = parser.get("aws_creds", "username")
password = parser.get("aws_creds", "password")
host = parser.get("aws_creds", "host")
port = parser.get("aws_creds", "port")

# Подключение к кластеру Redshift
rs_conn = psycopg2.connect(
    "dbname=" + dbname
    + " user=" + user
    + " password=" + password
    + " host=" + host
    + " port=" + port)
```

Теперь вы можете выполнить команду `COPY`, используя объект `Cursor psycopg2`. Выполните ту же команду `COPY`, которую вы запускали вручную ранее в этом разделе, но вместо жесткого кодирования идентификатора учетной записи AWS и имени роли IAM загрузите эти значения из файла pipeline.conf:

```
# Загрузить account_id и iam_role из файлов конфигурации
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
account_id = parser.get("aws_boto_credentials", "account_id")
iam_role = parser.get("aws_creds", "iam_role")
bucket_name = parser.get("aws_boto_credentials", "bucket_name")

# Запуск команды COPY для загрузки файла в Redshift
file_path = ("s3://"
    + bucket_name
    + "/order_extract.csv")
role_string = ("arn:aws:iam::"
    + account_id
    + ":role/" + iam_role)
```

```
sql = "COPY public.Orders"
sql = sql + " from %s "
sql = sql + " iam_role %s;"

# Создание объекта cursor и выполнение COPY
cur = rs_conn.cursor()
cur.execute(sql,(file_path, role_string))

# Закрытие объекта cursor и завершение транзакции
cur.close()
rs_conn.commit()

# Закрытие соединения
rs_conn.close()
```

Если целевая таблица не существует, то перед запуском скрипта ее нужно создать. В этом примере я загружаю данные, которые были извлечены в файл order_extract.csv (см. *раздел "Полное или инкрементное извлечение таблицы MySQL" главы 4*). Конечно, вы можете загрузить любые данные, какие захотите. Просто убедитесь, что целевая таблица имеет подходящую структуру. Чтобы создать целевую таблицу в вашем кластере, запустите следующий SQL-запрос через редактор запросов Redshift или другое приложение, подключенное к вашему кластеру:

```
CREATE TABLE public.Orders (
  OrderId int,
  OrderStatus varchar(30),
  LastUpdated timestamp
);
```

Наконец, запустите скрипт следующим образом:

```
(env) $ python copy_to_redshift.py
```

Инкрементные и полные загрузки

В предыдущем примере кода команда `COPY` загружала данные из извлеченного CSV-файла непосредственно в таблицу в кластере Redshift. Если данные в файле формата CSV получены инкрементным извлечением неизменного источника (как в случае с чем-то вроде неизменяемых данных событий или другого набо-

ра данных "только для вставки"), то больше ничего не нужно делать. Однако, если данные в CSV-файле содержат обновленные записи, а также вставки или все содержимое исходной таблицы, вам придется проделать немного больше работы или, по крайней мере, иметь ее в виду.

Возьмем случай с таблицей `Orders` из *раздела "Полное или инкрементное извлечение таблицы MySQL" главы 4*. Данные, которые вы загружаете из CSV-файла, были извлечены полностью или инкрементно из исходной таблицы MySQL.

Если данные были извлечены полностью, то нужно сделать одно небольшое дополнение к скрипту загрузки. Очистите целевую таблицу в Redshift (выполнив команду `TRUNCATE`) перед запуском операции `COPY`. Обновленный фрагмент кода выглядит так:

```
import boto3
import configparser
import psycopg2

parser = configparser.ConfigParser()
parser.read("pipeline.conf")
dbname = parser.get("aws_creds", "database")
user = parser.get("aws_creds", "username")
password = parser.get("aws_creds", "password")
host = parser.get("aws_creds", "host")
port = parser.get("aws_creds", "port")

# Подключение к кластеру redshift
rs_conn = psycopg2.connect(
    "dbname=" + dbname
    + " user=" + user
    + " password=" + password
    + " host=" + host
    + " port=" + port)

parser = configparser.ConfigParser()
parser.read("pipeline.conf")
account_id = parser.get("aws_boto_credentials", "account_id")
iam_role = parser.get("aws_creds", "iam_role")
bucket_name = parser.get("aws_boto_credentials", "bucket_name")
```

```
# Очистка целевой таблицы
sql = "TRUNCATE public.Orders;"
cur = rs_conn.cursor()
cur.execute(sql)

cur.close()
rs_conn.commit()

# Запуск команды COPY для загрузки файла в Redshift
file_path = ("s3://"
    + bucket_name
    + "/order_extract.csv")
role_string = ("arn:aws:iam::"
    + account_id
    + ":role/" + iam_role)

sql = "COPY public.Orders"
sql = sql + " from %s "
sql = sql + " iam_role %s;"

# Создание объекта cursor и выполнение команды COPY
cur = rs_conn.cursor()
cur.execute(sql,(file_path, role_string))

# Закрытие объекта cursor курсор и транзакции
cur.close()
rs_conn.commit()

# Закрытие соединения
rs_conn.close()
```

При инкрементном извлечении данных очищать целевую таблицу не нужно. Если вы это сделаете, все, что у вас останется, — это обновленные записи последнего запуска задания извлечения. Есть несколько способов обработки данных, извлеченных таким образом, но лучше не усложнять процесс без необходимости.

В этом случае вы можете просто загрузить данные с помощью команды `COPY` (без `TRUNCATE`!) и полагаться на метку времени, указывающую, когда запись была в последний раз обновлена, чтобы позже определить, какая запись является последней, или вернуться к исторической записи. Например, предположим, что запись

в исходной таблице была изменена и, следовательно, присутствует в загружаемом CSV-файле. После загрузки вы увидите что-то вроде содержимого табл. 5.1 в таблице назначения Redshift.

Таблица 5.1. *Содержимое таблицы* `Orders` *в Redshift*

OrderId	OrderStatus	LastUpdated
1	Размещен	2020-06-01 12:00:00
1	Отгружен	2020-06-09 12:00:25

Как видно из табл. 5.1, заказ со значением идентификатора 1 присутствует в таблице дважды. Первая запись существовала до последней загрузки, а вторая была только что загружена из CSV-файла. Первая запись поступила из-за обновления записи 1 июня 2020 г., когда заказ находился в состоянии "Размещен". Второе обновление состоялось 9 июня 2020 г., когда заказ был отгружен, и включено в последний загруженный вами CSV-файл.

С точки зрения ведения исторических записей идеально было бы иметь обе эти записи в целевой таблице. Позже, на этапе преобразования конвейера, аналитик может выбрать одну или обе записи в зависимости от потребностей конкретного анализа. Возможно, он захочет узнать, как долго заказ находился в состоянии ожидания. Для этого понадобятся обе записи. Если он захочет узнать текущий статус заказа, эта информация тоже окажется доступной.

Хотя может показаться неудобным иметь несколько записей для одного и того же `OrderId` в целевой таблице, в данном случае это правильно! Цель этапа сбора данных — сосредоточиться на извлечении и загрузке данных. Что делать с данными — это задача этапа преобразования конвейера, о котором пойдет речь в *главе 6*.

Загрузка данных, извлеченных из журнала CDC

Если ваши данные были извлечены с помощью метода CDC, нужно отметить еще одно соображение. Хотя это похоже на загрузку данных, которые были извлечены инкрементно, у вас бу-

дет доступ не только к вставленным и обновленным, но и к удаленным записям.

Возьмем пример извлечения двоичного журнала MySQL из *главы 4*. Вспомним, что результатом выполнения примера кода был CSV-файл с именем orders_extract.csv, который мы загрузили в корзину S3. Его содержимое выглядело следующим образом:

```
insert|1|Размещен|2020-06-01 12:00:00
update|1|Отгружен|2020-06-09 12:00:25
```

Как и в примере с инкрементной загрузкой, рассмотренном ранее, здесь есть две записи для `OrderId 1`. При загрузке в хранилище данные выглядят так, как в табл. 5.1. Но, в отличие от предыдущего примера, orders_extract.csv содержит столбец для события, отвечающего за запись в файле. В этом примере это либо `insert`, либо `update`. Если бы других типов событий не было, вы могли бы проигнорировать поле события и в итоге получить таблицу в Redshift, похожую на табл. 5.1. Оттуда аналитики будут иметь доступ к обеим записям, когда они позже начнут строить модели данных в конвейере. Однако рассмотрим другую версию orders_extract.csv, в которую включена еще одна строка:

```
insert|1|Размещен|2020-06-01 12:00:00
update|1|Отгружен|2020-06-09 12:00:25
delete|1|Отгружен|2020-06-10 9:05:12
```

Третья строка показывает, что запись о заказе была удалена на следующий день после ее обновления. При полном извлечении запись полностью исчезнет, а инкрементное извлечение не подхватит удаление (см. *раздел "Извлечение данных из БД MySQL" главы 4* для получения более подробной информации). Однако благодаря использованию CDC событие удаления было подхвачено и включено в CSV-файл.

Чтобы разместить удаленные записи, необходимо добавить в целевую таблицу в хранилище Redshift столбец для хранения типа события. В табл. 5.2 показано, как выглядит расширенная версия `Orders`.

И снова отметим, что цель этапа сбора данных в конвейер — эффективное извлечение данных из источника и их загрузка в место назначения. Шаг преобразования в конвейере — это место, где находится логика моделирования данных для конкретного вари-

анта использования. В *главе 6* обсуждается, как моделировать данные, загруженные с помощью сбора CDC, как в этом примере.

Таблица 5.2. *Таблица* `Orders` *со столбцом* `EventType` *в Redshift*

EventType	OrderId	OrderStatus	LastUpdated
insert	1	Размещен	2020-06-01 12:00:00
update	1	Отгружен	2020-06-09 12:00:25
delete	1	Отгружен	2020-06-10 9:05:12

Настройка хранилища Snowflake в качестве пункта назначения

Если хранилищем данных у вас служит Snowflake, т. е. три варианта настройки доступа к корзине S3 из вашего экземпляра Snowflake:

- ❒ настроить интеграцию хранилища Snowflake;
- ❒ настроить роль AWS IAM;
- ❒ настроить пользователя AWS IAM.

Из этих трех вариантов рекомендуется первый из-за беспрепятственного использования интеграции хранилища Snowflake при последующем взаимодействии с корзиной S3 от Snowflake. Поскольку настройка включает в себя ряд специфических шагов, лучше всего обратиться к документации Snowflake по этой теме.

На последнем этапе настройки вы создадите *внешнюю площадку* (external stage). Внешняя площадка — это объект, который указывает на внешнее хранилище, поэтому Snowflake может получить к нему доступ. У нас внешним хранилищем будет служить корзина S3, которую вы создали ранее.

Прежде чем создавать площадку, удобно определить параметр `FILE FORMAT` в Snowflake, который понадобится вам для площадки и позже — для файлов аналогичных форматов. Поскольку примеры в этой главе формируют CSV-файлы с разделением вертикальной чертой, создайте следующий `FILE FORMAT`:

```
CREATE or REPLACE FILE FORMAT pipe_csv_format
TYPE = 'csv'
FIELD_DELIMITER = '|';
```

При создании площадки для корзины S3 в соответствии с последним шагом документации Snowflake синтаксис будет выглядеть примерно так:

```
USE SCHEMA my_db.my_schema;

CREATE STAGE my_s3_stage
  storage_integration = s3_int
  url = 's3://pipeline-bucket/'
  file_format = pipe_csv_format;
```

В *разделе "Загрузка данных в хранилище Snowflake"* этой главы вы будете использовать площадку для загрузки в Snowflake данных, которые были извлечены и сохранены в корзине S3.

Наконец, вам нужно добавить в файл pipeline.conf раздел с учетными данными для входа в Snowflake. Обратите внимание, что указанный вами пользователь должен иметь разрешение `USAGE` на только что созданную вами площадку. Кроме того, значение `account_name` должно быть отформатировано в зависимости от вашего облачного провайдера и региона, в котором находится учетная запись. Например, если ваша учетная запись называется `snowflake_acct1` и размещена в регионе AWS US East (Ohio), то значением `account_name` будет `snowflake_acct1.us-east-2.aws`. Поскольку это значение потребуется для подключения к Snowflake через Python с помощью библиотеки snowflake-connector-python, я рекомендую обратиться к документации библиотеки, чтобы определить правильное значение для вашего `account_name`.

Вот раздел, который нужно добавить в pipeline.conf:

```
[snowflake_creds]
username = snowflake_user
password = snowflake_password
account_name = snowflake_acct1.us-east-2.aws
```

Загрузка данных в хранилище Snowflake

Загрузка данных в Snowflake выполняется почти так же, как и в предыдущих разделах о загрузке данных в Redshift. Поэтому я не буду обсуждать особенности передачи данных, полученных путем полного, инкрементного или CDC-извлечения. Здесь я опишу только синтаксис загрузки данных из извлеченного файла.

Механизм загрузки данных в Snowflake — команда `COPY INTO`, которая загружает содержимое файла или нескольких файлов в таблицу в хранилище Snowflake. Вы можете узнать больше о расширенном использовании и параметрах команды в документации Snowflake.

> **ПРИМЕЧАНИЕ**
>
> У Snowflake также есть служба интеграции данных под названием Snowpipe, которая позволяет загружать данные из файлов, как только они становятся доступными на этапе Snowflake, подобном файлу, который рассмотрен в примере этого раздела. С помощью Snowpipe вы можете непрерывно загружать данные вместо планирования массовой загрузки командой `COPY INTO`.

В каждом примере извлечения в *главе 4* файл CSV записывался в корзину S3. В *разделе "Настройка хранилища Snowflake в качестве места назначения"* вы создали площадку Snowflake с именем my_s3_stage, которая связана с этим сегментом. Теперь с помощью команды `COPY INTO` вы можете загрузить файл в таблицу Snowflake следующим образом:

```
COPY INTO destination_table
  FROM @my_s3_stage/extract_file.csv;
```

Также можно загрузить в таблицу несколько файлов одновременно. В некоторых случаях данные извлекаются более чем в один файл из-за большого объема или в результате выполнения нескольких заданий извлечения с момента последней загрузки. Если файлы имеют однообразный шаблон именования (а они должны!), вы можете загрузить их все, задав параметр `pattern`:

```
COPY INTO destination_table
  FROM @my_s3_stage
  pattern='.*extract.*.csv';
```

ПРИМЕЧАНИЕ

Формат загружаемого файла был задан при создании площадки Snowflake (CSV-файл с разделением вертикальной чертой), поэтому вам не нужно указывать его в синтаксисе команды `COPY INTO`.

Теперь, когда вы знаете, как работает команда `COPY INTO`, пришло время написать короткий скрипт Python, который можно запланировать и выполнить для автоматизации загрузки конвейера. Дополнительные сведения об этом и о других методах оркестровки конвейера см. в *главе 7*.

Сначала вам нужно установить библиотеку Python для подключения к вашему экземпляру Snowflake. Вы можете сделать это с помощью `pip`:

```
(env) $ pip install snowflake-connector-python
```

Теперь можно написать простой скрипт Python для подключения к вашему экземпляру Snowflake и командой `COPY INTO` загружать содержимое CSV-файла в целевую таблицу:

```
import snowflake.connector
import configparser

parser = configparser.ConfigParser()
parser.read("pipeline.conf")
username = parser.get("snowflake_creds",
            "username")
password = parser.get("snowflake_creds",
            "password")
account_name = parser.get("snowflake_creds",
            "account_name")

snow_conn = snowflake.connector.connect(
    user = username,
    password = password,
    account = account_name
    )

sql = """COPY INTO destination_table
  FROM @my_s3_stage
  pattern='.*extract.*.csv';"""
```

```
cur = snow_conn.cursor()
cur.execute(sql)
cur.close()
```

Использование вашего файлового хранилища в качестве озера данных

Бывают случаи, когда имеет смысл извлекать данные из корзины S3 (или другого облачного хранилища), а не загружать в хранилище данных. Данные, хранящиеся в структурированной или частично структурированной форме, часто называют *озером данных* (data lake).

В отличие от хранилища, озеро данных хранит данные во многих форматах в необработанном и иногда неструктурированном виде. Их дешевле хранить, но они не оптимизированы для запросов так же, как структурированные данные в хранилище.

Однако в последние годы появились инструменты, делающие запросы к данным в озере гораздо более доступными и часто прозрачными для пользователя, которому удобно работать с SQL. Например, Amazon Athena — это сервис AWS, который позволяет пользователю запрашивать данные, хранящиеся в S3, с помощью SQL. Amazon Redshift Spectrum — это сервис, который позволяет Redshift получать доступ к данным в S3 как к внешней таблице и ссылаться на нее в запросах вместе с таблицами в хранилище Redshift. Другие поставщики облачных услуг и продукты имеют аналогичную функциональность.

Когда следует задуматься о таком подходе вместо структурирования и загрузки данных в хранилище? Есть несколько характерных ситуаций.

Хранение больших объемов данных в озере данных на основе облачного хранилища обходится дешевле, чем в обычном хранилище (это не относится к озерам данных Snowflake, которые используют то же хранилище, что и хранилища данных Snowflake). Кроме того, поскольку это неструктурированные или частично структурированные данные (без предопределенной схемы), вносить изменения в типы или свойства хранимых данных намного проще, чем менять схемы хранилища. Документы JSON — это

пример полуструктурированных данных, с которыми вы можете столкнуться в озере данных. Если структура данных часто меняется, то имеет смысл рассмотреть возможность ее хранения в озере данных, по крайней мере, в определенный период времени.

На этапе исследования данных или проекта машинного обучения специалист по данным или инженер по машинному обучению может еще не знать точно, в какой "форме" нужны данные. Получив доступ к данным в озере в необработанном виде, они могут исследовать данные и определить, какие атрибуты данных им необходимы. Как только они разберутся в данных, можно принять решение, имеет ли смысл загружать данные в таблицу в хранилище, и оптимизировать соответствующие запросы.

На практике многие организации имеют в своей инфраструктуре данных как озера, так и хранилища данных. Со временем эти два решения стали взаимодополняющими, а не конкурирующими.

Фреймворки с открытым исходным кодом

Как вы уже заметили, при каждом сборе данных есть повторяющиеся шаги (как при извлечении, так и при загрузке). Поэтому в последние годы появилось множество фреймворков, которые обеспечивают основные функциональные возможности и соединения с общими источниками и пунктами назначения данных. Некоторые из них имеют открытый исходный код, как описано в этом разделе, а в следующем разделе представлен обзор некоторых популярных коммерческих продуктов для сбора данных.

Один из популярных фреймворков с открытым исходным кодом — Singer. Написанный на Python, Singer использует *отводы* (tap, краник) для извлечения данных из источника и потоковой передачи их в формате JSON к *цели* (target). Например, если вы хотите извлечь данные из БД MySQL и загрузить их в хранилище данных Google BigQuery, то у вас должен быть отвод MySQL и цель BigQuery.

Как и в случае с примерами кода в этой главе, с Singer вам все равно понадобится отдельная структура оркестровки для планирования и координации приема данных (дополнительную инфор-

мацию см. в *главе 7*). Однако независимо от того, применяете ли вы Singer или другой фреймворк, надежный фундамент поможет вам быстро приступить к работе.

Поскольку это проект с открытым исходным кодом, доступно большое количество отводов и целей (некоторые из наиболее популярных перечислены в табл. 5.3), и вы также можете внести свой вклад в проект. Singer хорошо задокументирован и имеет активные сообщества Slack и GitHub.

Таблица 5.3. *Популярные модули отводов и целей фреймворка Singer*

Отводы	Цели
Google Analytics	CSV
Jira	Google BigQuery
MySQL	PostgreSQL
PostgreSQL	Amazon Redshift
Salesforce	Snowflake

Коммерческие альтернативы

Существует несколько облачных коммерческих продуктов, которые позволяют организовать сбор многих распространенных типов данных без написания программного кода. У них также есть встроенные планировщики и оркестраторы. Конечно, все это стоит денег.

Двумя наиболее популярными коммерческими инструментами для сбора данных являются Stitch и Fivetran. Оба полностью основаны на веб-интерфейсе и доступны для инженеров данных, а также для других специалистов по данным в группе обработки и анализа данных. Они предоставляют сотни готовых коннекторов для популярных источников данных, таких как Salesforce, HubSpot, Google Analytics, GitHub и других. Вы также можете получать данные из MySQL, Postgres и других баз данных. Имеется встроенная поддержка Amazon Redshift, Snowflake и других хранилищ данных.

Если вы собираете данные из поддерживаемых источников, то сэкономите много времени при создании нового сборщика дан-

ных. Кроме того, как подробно описано в *главе 7*, планирование и организация сбора данных — нетривиальные задачи. С помощью Stitch и Fivetran вы сможете создавать, планировать и отслеживать конвейеры сбора данных прямо в браузере.

Некоторые коннекторы на обеих платформах также поддерживают тайм-ауты выполнения заданий, обработку повторяющихся данных, изменение схемы исходной системы и многое другое. Если вы создаете сборщик данных самостоятельно, то вам придется реализовать все эти опции самостоятельно.

Конечно, у коммерческих решений есть не только плюсы, но и минусы:

- Расходы.

  И у Stitch, и у Fivetran есть модели ценообразования, основанные на объеме. Хотя они различаются тем, как измеряют объем и какие функции включают в каждую ценовую категорию, в целом ваши расходы зависят от того, сколько данных вы получаете. Если у вас есть несколько источников данных большого объема, из которых вы регулярно получаете данные, это может привести к большим расходам.

- Привязка к поставщику.

  После того как вы потратите значительные средства на выбранного поставщика, вам придется столкнуться с нетривиальным объемом работы по переходу на другой инструмент или продукт, если вы решите двигаться дальше в будущем.

- Настройка требует программирования.

  Если исходная система, из которой вы хотите получить данные, не имеет предварительно созданного коннектора, вам придется написать небольшой объем кода самостоятельно. Для Stitch это означает написание пользовательского отвода Singer (см. предыдущий раздел), а для Fivetran вам нужно будет написать облачные функции, используя AWS Lambda, Azure Function или Google Cloud Functions. Если у вас много пользовательских источников данных, таких как настраиваемые REST API, то в конечном итоге вам придется писать собственный программный код, а затем платить за Stitch или Fivetran для его запуска.

- Безопасность и конфиденциальность.

  Хотя оба продукта служат для передачи ваших данных и не хранят их в течение длительного периода времени, технически они по-прежнему имеют доступ как к вашим исходным системам, так и к местам назначения (обычно к хранилищам данных или озерам данных). И Fivetran, и Stitch соответствуют высоким стандартам безопасности, однако некоторые организации неохотно используют их из-за строгих нормативов оценки рисков, потенциальной ответственности и накладных расходов, связанных с рассмотрением и утверждением нового процессора данных.

Выбор "создать или купить" сложен и уникален для каждой организации и варианта применения. Также следует иметь в виду, что некоторые организации для сбора данных сочетают пользовательский код и такие продукты, как Fivetran или Stitch. Например, может быть целесообразно написать собственный код для обработки некоторых больших объемов загрузки, запуск которых на коммерческой платформе был бы дорогостоящим, но при этом использовать Stitch или Fivetran для загрузки с готовыми коннекторами, поддерживаемыми поставщиком.

Если вы выберете сочетание пользовательских и коммерческих инструментов, имейте в виду, что вам нужно подумать о том, как вы стандартизируете такие вещи, как ведение журнала, оповещение и управление зависимостями. В последующих главах этой книги обсуждаются данные темы и затрагиваются проблемы управления конвейерами, охватывающими несколько платформ.

ГЛАВА 6

Преобразование данных

В шаблоне ELT (см. *главу 3*) после приема данных в озеро или хранилище (см. *главу 4*) следующим шагом конвейера является преобразование данных, которое может включать как неконтекстную манипуляцию данными, так и моделирование данных с учетом бизнес-контекста и логики.

Если конвейер предназначен для представления или анализа данных, то в дополнение к любым неконтекстным преобразованиям данные дополнительно преобразуются в модели. В *главе 2* мы выяснили, что модель данных структурирует и определяет данные в формате, понятном и оптимизированном для анализа. Модель данных представлена в виде одной или нескольких таблиц в хранилище данных.

Хотя созданием неконтекстных преобразований в конвейере иногда занимаются инженеры данных, в настоящее время подавляющее большинство преобразований выполняют аналитики данных и инженеры-аналитики. У людей на этих должностях больше возможностей, чем когда-либо, благодаря появлению шаблона ELT (у них есть данные, которые им нужны, прямо в хранилище!), а также вспомогательных инструментов и платформ, разработанных с использованием SQL в качестве основного языка.

В этой главе рассматриваются как неконтекстные преобразования, которые являются общими почти для каждого конвейера данных, так и модели данных, которые предназначены для информационных панелей, отчетов и однократного анализа бизнес-проблемы. Поскольку SQL — это язык аналитика данных и инженера-аналитика, большинство примеров кода преобразования написано на SQL. Я включил в эту главу несколько примеров, написанных на Python, чтобы проиллюстрировать, когда имеет

смысл тесно связывать неконтекстные преобразования с приемом данных, применяя мощные библиотеки Python.

Как и в случаях сбора данных, рассмотренных в *главах 4* и *5*, примеры кода сильно упрощены и служат в качестве отправной точки для более сложных преобразований. Про управление зависимостями между преобразованиями и другими этапами конвейера будет рассказано в *главе 8*.

SQL-совместимость

SQL-запросы в этой главе совместимы с большинством диалектов SQL. В них редко встречается ограниченный синтаксис, зависящий от поставщика, поэтому они должны работать с любой современной базой данных, поддерживающей SQL (с небольшими изменениями или в неизменном виде).

Неконтекстные преобразования

В *главе 3* я кратко упомянул о существовании подшаблона EtLT, где буква *t* в нижнем регистре означает некоторые неконтекстные преобразования данных, такие как:

- удаление дубликатов записей в таблице;
- разделение параметров URL на отдельные компоненты.

Хотя существует бесчисленное множество вариантов, предоставляя образцы кода для двух упомянутых операций, я надеюсь охватить наиболее распространенные случаи неконтекстных преобразований. В следующем разделе рассказывается о том, когда имеет смысл выполнять такие преобразования в процессе приема данных (EtLT), а когда — после приема (ELT).

Удаление дубликатов записей в таблице

Мир не идеален, и в таблице данных, которые были загружены в хранилище, могут существовать повторяющиеся записи. Это происходит по ряду причин:

- сбор добавочных данных ошибочно перекрывает предыдущее временное окно сбора и извлекает записи, которые уже были загружены при предыдущем запуске;

- в исходной системе были случайно созданы повторяющиеся записи;
- данные, которые были внесены ранее, перекрывались с последующими данными, загруженными в таблицу во время сбора.

Какой бы ни была причина, проверку и удаление повторяющихся записей лучше всего выполнять с помощью SQL-запросов. Каждый из представленных далее SQL-запросов обращается к таблице `Orders` в БД, показанной в табл. 6.1. Таблица содержит пять записей, две из которых дублированы. Хотя существуют три записи для `OrderId 1`, вторая и четвертая строки абсолютно одинаковы. Цель рассматриваемого примера — определить это дублирование и устранить его. Несмотря на то что в этом примере есть только две абсолютно одинаковые записи, логика в следующих примерах кода работает аналогично, если в таблице есть три, четыре или даже больше копий одной и той же записи.

Таблица 6.1. *Таблица* `Orders` *с дубликатами записей*

OrderId	OrderStatus	LastUpdated
1	Размещен	2020-06-01
1	Отгружен	2020-06-09
2	Отгружен	2020-07-11
1	Отгружен	2020-06-09
3	Отгружен	2020-07-12

Если вы хотите создать заполняемую таблицу `Orders` для использования в примерах, приведенных в листингах 6.1 и 6.2, воспользуйтесь следующим SQL-запросом:

```
CREATE TABLE Orders (
  OrderId int,
  OrderStatus varchar(30),
  LastUpdated timestamp
);

INSERT INTO Orders VALUES(1,'Размещен', '2020-06-01');
INSERT INTO Orders   VALUES(1,'Отгружен', '2020-06-09');
INSERT INTO Orders VALUES(2,'Отгружен', '2020-07-11');
```

```
INSERT INTO Orders VALUES(1,'Отгружен', '2020-06-09');
INSERT INTO Orders VALUES(3,'Отгружен', '2020-07-12');
```

Выявить повторяющиеся записи в таблице очень просто с помощью SQL-операторов `GROUP BY` и `HAVING`. Следующий запрос возвращает все повторяющиеся записи вместе с подсчетом их числа:

```
SELECT OrderId,
  OrderStatus,
  LastUpdated,
  COUNT(*) AS dup_count
FROM Orders
GROUP BY OrderId, OrderStatus, LastUpdated
HAVING COUNT(*) > 1;
```

После запуска запрос возвращает следующую информацию:

```
OrderId | OrderStatus | LastUpdated | dup_count
1       | Отгружен    | 2020-06-09  | 2
```

Теперь, когда вы знаете, что существует по крайней мере один дубликат, вы можете удалить повторяющиеся записи. Я собираюсь рассказать о двух способах сделать это. Выбор метода зависит от многих факторов, связанных с оптимизацией вашей БД, а также от ваших предпочтений в синтаксисе SQL. Я предлагаю попробовать оба метода и сравнить время выполнения.

Первый метод заключается в использовании последовательности запросов. Первый запрос создает копию исходной таблицы с помощью оператора `DISTINCT`. В результате выполнения первого запроса формируется набор данных только из четырех строк, т. к. две повторяющиеся строки превращаются в одну благодаря `DISTINCT`. Далее исходная таблица очищается. Наконец, дедуплицированная версия набора данных вставляется в исходную таблицу, как показано в листинге 6.1.

Листинг 6.1. Запрос distinct_orders_1.sql

```
CREATE TABLE distinct_orders AS
SELECT DISTINCT OrderId,
  OrderStatus,
  LastUpdated
FROM ORDERS;
```

```
TRUNCATE TABLE Orders;

INSERT INTO Orders
SELECT * FROM distinct_orders;

DROP TABLE distinct_orders;
```

ПРЕДУПРЕЖДЕНИЕ

После операции `TRUNCATE` таблица `Orders` останется пустой до тех пор, пока не будет завершена следующая операция `INSERT`. Все это время таблица `Orders` пуста и практически недоступна ни одному пользователю или процессу, который ее запрашивает. Хотя операция `INSERT` обычно не занимает много времени, в случае очень больших таблиц вы можете удалить таблицу `Orders`, а затем переименовать `distinct_orders` в `Orders`.

Другой подход заключается в применении *оконной функции* (window function) для группировки повторяющихся строк и присвоения им номеров строк, чтобы определить, какие из них следует удалить, а какие оставить. Я буду использовать функцию `ROW_NUMBER` для ранжирования записей и оператор `PARTITION BY` для группировки записей по каждому столбцу. При этом любой группе записей с более чем одним совпадением (наши дубликаты) будет присвоено значение `ROW_NUMBER` больше единицы.

Если вы выполнили пример из листинга 6.1, то обязательно обновите таблицу `Orders` с помощью операторов `INSERT`, приведенных ранее в этом разделе, чтобы она снова содержала исходные данные, показанные в табл. 6.1. Для работы со следующим примером вам снова понадобятся повторяющиеся строки!

Вот что происходит, когда такой запрос выполняется для таблицы `Orders`:

```
SELECT OrderId,
  OrderStatus,
  LastUpdated,
  ROW_NUMBER() OVER(PARTITION BY OrderId,
                    OrderStatus,
                    LastUpdated)
    AS dup_count
FROM Orders;
```

Запрос вернет следующий результат:

```
orderid | orderstatus  | lastupdated | dup_count
--------+--------------+-------------+----------
      1 | Размещен     | 2020-06-01  |       1
      1 | Отгружен     | 2020-06-09  |       1
      1 | Отгружен     | 2020-06-09  |       2
      2 | Отгружен     | 2020-07-11  |       1
      3 | Отгружен     | 2020-07-12  |       1
```

Как видите, третья строка в результирующем наборе имеет значение `dup_count`, равное `2`, поскольку она является дубликатом записи прямо над ней. Теперь, как и в первом подходе, вы можете создать таблицу с дедуплицированными записями, очистить таблицу `Orders` и, наконец, вставить очищенный набор данных в `Orders`. Листинг 6.2 содержит полный исходный код.

Листинг 6.2. Код SQL-скрипта distinct_orders_2.sql

```
CREATE TABLE all_orders AS
SELECT
  OrderId,
  OrderStatus,
  LastUpdated,
  ROW_NUMBER() OVER(PARTITION BY OrderId,
                    OrderStatus,
                    LastUpdated)
    AS dup_count
FROM Orders;

TRUNCATE TABLE Orders;

-- Вставка только уникальных записей
INSERT INTO Orders (OrderId, OrderStatus, LastUpdated)
SELECT
  OrderId,
  OrderStatus,
  LastUpdated
FROM all_orders
WHERE
  dup_count = 1;

DROP TABLE all_orders;
```

Независимо от того, какой подход вы выберете, результатом будет дедуплицированная версия таблицы `Orders`, как показано в табл. 6.2.

Таблица 6.2. *Таблица Orders без дубликатов*

OrderId	OrderStatus	LastUpdated
1	Размещен	2020-06-01
1	Отгружен	2020-06-09
2	Отгружен	2020-07-11
3	Отгружен	2020-07-12

Парсинг URL-адресов

Анализ сегментов URL-адресов — это задача, практически не связанная с бизнес-контекстом. Существует ряд компонентов URL, которые можно проанализировать на этапе преобразования и сохранить в отдельных столбцах таблицы базы данных.

Например, рассмотрим следующий URL-адрес:

https://www.mydomain.com/page-name?utm_content=textlink&utm_medium=social&utm_source=twitter&utm_campaign=fallsale

Эта строка содержит шесть компонентов, которые представляют собой полезные данные и могут быть проанализированы и сохранены в отдельных столбцах:

- домен: *www.domain.com*;
- путь URL: /page-name;
- значение параметра utm_content: *textlink*;
- значение параметра utm_medium: *social*;
- значение параметра utm_source: *twitter*;
- значение параметра utm_campaign: *fallsale*.

UTM-параметры

Параметры Urchin Tracking Module (UTM) — это параметры URL, которые предназначены для отслеживания маркетинговых и рекламных кампаний. Они широко применяются на большинстве платформ и в организациях.

Для парсинга URL-адресов можно применить как SQL, так и Python. Вы можете выбрать инструмент исходя из времени запуска преобразования и места хранения URL-адресов. Например, если вы следуете шаблону EtLT и можете анализировать URL-адреса после извлечения из источника, но перед загрузкой в таблицу в хранилище данных, отличным выбором будет Python. Я начну с примера на Python, а затем перейду на SQL.

Сначала установите библиотеку Python urllib3 с помощью команды `pip`:

```
(env) $ pip install urllib3
```

Инструкции по настройке Python приведены в *разделе "Настройка среды Python" главы 4.*

Затем используйте функции `urlsplit` и `parse_qs` для анализа соответствующих компонентов URL-адреса. В следующем примере кода я делаю это и вывожу результаты на печать:

```
from urllib.parse import urlsplit, parse_qs

url = """https://www.mydomain.com/page-name?utm_content=textlink&utm_
medium=social&utm_source=twitter&utm_campaign=fallsale"""

split_url = urlsplit(url)
params = parse_qs(split_url.query)

# домен
print(split_url.netloc)

# путь url
print(split_url.path)

# параметры utm
print(params['utm_content'][0])
print(params['utm_medium'][0])
print(params['utm_source'][0])
print(params['utm_campaign'][0])
```

После запуска этот пример кода выдает следующее:

```
www.mydomain.com
/page-name
textlink
```

```
social
twitter
fallsale
```

Как и в примерах кода приема данных из *глав 4* и *5*, вы также можете проанализировать и записать каждый параметр в CSV-файл, чтобы загрузить его в хранилище для завершения сбора данных. Листинг 6.3 содержит пример кода, который делает такую операцию для единственного образца URL-адреса, но вы, вероятно, будете перебирать более одного URL-адреса!

Листинг 6.3. Пример скрипта url_parse.sql

```
from urllib.parse import urlsplit, parse_qs
import csv

url = """https://www.mydomain.com/page-name?utm_content=textlink&utm_
medium=social&utm_source=twitter&utm_campaign=fallsale"""

split_url = urlsplit(url)
params = parse_qs(split_url.query)
parsed_url = []
all_urls = []

# домен
parsed_url.append(split_url.netloc)

# путь url
parsed_url.append(split_url.path)
parsed_url.append(params['utm_content'][0])
parsed_url.append(params['utm_medium'][0])
parsed_url.append(params['utm_source'][0])
parsed_url.append(params['utm_campaign'][0])

all_urls.append(parsed_url)

export_file = "export_file.csv"

with open(export_file, 'w') as fp:
        csvw = csv.writer(fp, delimiter='|')
        csvw.writerows(all_urls)
fp.close()
```

Если вам нужно проанализировать URL-адреса, которые уже были загружены в хранилище данных с помощью SQL, это может оказаться более сложной задачей. Хотя некоторые поставщики хранилищ данных предоставляют функции для анализа URL-адресов, но так бывает не всегда. Snowflake, например, предоставляет функцию PARSE_URL, которая разбирает URL-адрес на компоненты и возвращает результат в виде объекта JSON. Например, если вы хотите проанализировать URL-адрес из предыдущего примера, результат будет выглядеть следующим образом:

```
SELECT parse_url('https://www.mydomain.com/pagename?utm_content=
textlink&utm_medium=social&utm_source=twitter&utm_campaign=
fallsale');
```

```
+-------------------------------------------------+
| PARSE_URL('https://www.mydomain.com/page-name?
utm_content=textlink&utm_medium=social&utm_source=
twitter&utm_campaign=fallsale')                   |
|-------------------------------------------------|
| {                                               |
| "fragment": null,                               |
| "host": "www.mydomain.com",                     |
| "parameters": {                                 |
| "utm_content": "textlink",                      |
| "utm_medium": "social",                         |
| "utm_source": "twitter",                        |
| "utm_campaign": "fallsale"                      |
| },                                              |
| "path": "/page-name",                           |
| "query":
"utm_content=textlink&utm_medium=social&utm_source=
twitter&utm_campaign=fallsale",                   |
| "scheme": "HTTPS"                               |
| }                                               |
+-------------------------------------------------+
```

Если вы применяете Redshift или другую платформу хранилища данных без встроенной функции парсинга URL, то вам потребуется использовать собственный код или регулярные выражения. Например, в Redshift есть функция REGEXP_SUBSTR. Учитывая сложность синтаксического анализа URL-адресов в большинстве хранилищ данных, я рекомендую выполнять такой анализ с помо-

щью Python или другого языка во время приема данных и загрузки в структурированные компоненты URL-адресов.

Сохраните исходный URL!

Независимо от того, анализируете ли вы URL-адреса во время приема данных или после него, лучше также сохранить исходную строку URL-адреса в хранилище данных. URL-адреса могут иметь ряд параметров, которые вы не планировали анализировать и структурировать, но они могут пригодиться в будущем.

Когда лучше выполнять преобразование?

Преобразования данных, не связанные с бизнес-контекстом, подобные рассмотренным в предыдущем разделе, с технической точки зрения можно выполнять либо в процессе сбора данных, либо после его окончания. Однако есть несколько причин, по которым вам следует рассматривать их как часть процесса сбора (шаблон EtLT):

1. *Преобразование, которое проще всего выполнить с помощью языка, отличного от SQL*: как и при анализе URL-адресов в предыдущем примере. Если вы обнаружите, что гораздо проще использовать для преобразования библиотеки Python, сделайте это частью сбора данных. В шаблоне ELT преобразования, выполняемые после сбора, ограничены моделированием данных, и занимаются этим аналитики данных, которые обычно лучше всего разбираются в SQL.
2. *Преобразование направлено на решение проблемы качества данных.* Лучше всего решать проблему качества данных как можно раньше в конвейере (более подробная информация по этой теме содержится в *главе 9*). Например, в предыдущем разделе я представил пример выявления и удаления повторяющихся записей, которые были загружены в хранилище. Не следует рисковать тем, что аналитик "споткнется" о дубликаты данных, если вы можете выявить их и исправить на этапе сбора. Несмотря на то что преобразование написано на SQL, его можно запустить в конце этапа сбора, а не ждать, пока аналитик сам преобразует данные.

Когда дело доходит до преобразований, связанных с бизнес-логикой, лучше отделить их от сбора данных. Как вы увидите

в следующем разделе, этот тип преобразования называется *моделированием данных*.

Основы моделирования данных

Моделирование данных для анализа, информационных панелей и отчетов — это тема, которой стоит посвятить целую книгу. Однако есть некоторые базовые принципы моделирования данных в шаблоне ELT, которые мы рассмотрим в этом разделе.

В отличие от предыдущего раздела, в соответствии с шаблоном ELT при моделировании данных бизнес-контекст учитывается на этапе преобразования. Модели данных отражают смысл всех данных, которые были загружены в хранилище из различных источников на этапах извлечения и загрузки (сбор данных).

Ключевые термины моделирования данных

Когда я употребляю в этом разделе термин "модели данных", я имею в виду отдельные таблицы SQL в хранилище данных. В примерах моделей данных я сосредоточусь на двух свойствах моделей:

- *Меры* — это показатели, которые вам нужно измерить, например число клиентов и величина дохода в долларах.
- *Атрибуты* — это показатели, по которым вы хотите отфильтровать или сгруппировать в отчете или панели мониторинга. Примерами атрибутов могут служить даты, имена клиентов и названия стран.

Кроме того, я буду говорить о степени детализации модели данных. *Детализация* (granularity) — это уровень разбиения данных, хранящихся в модели. Например, модель, которая должна сообщать число заказов, размещенных каждый день, потребует детализации по дням. Если бы нужно было ответить на вопрос, сколько заказов размещается каждый час, то потребовалась бы почасовая детализация.

Исходные таблицы — это таблицы, которые были загружены в хранилище или озеро данных посредством сбора данных, как описано в *главах 4* и *5*. При моделировании данных модели строятся как из исходных таблиц, так и из других моделей.

Повторное использование кода модели данных

Хотя каждая модель данных представлена собственной таблицей в хранилище данных, логика ее построения может опираться на другие модели данных. Дополнительные сведения о преимуществах повторного использования логики и получения одной модели из другой см. в *разделе "Повторное использование логики модели данных" главы 9*.

Моделирование полностью обновляемых данных

При моделировании данных, которые были полностью перезагружены, например, как описано в *разделе "Извлечение данных из БД MySQL" главы 4*, вы сталкиваетесь с таблицей (или несколькими таблицами), которые содержат последнее состояние исходного хранилища данных. Например, в табл. 6.3 показаны записи в таблице `Orders`, аналогичной табл. 6.2, но только с последними записями, а не с полной историей. Обратите внимание, что запись `Размещен` для `OrderId 1` в этой версии отсутствует. Ситуация напоминает полную загрузку таблицы из исходной БД в хранилище. Другими словами, она похожа на текущее состояние таблицы `Orders` в исходной системе на момент сбора данных.

Четвертый столбец `CustomerId`, в котором хранится идентификатор покупателя, разместившего заказ, и пятый столбец `OrderTotal`, содержащий стоимость заказа в долларах, в табл. 6.3 также отличаются от табл. 6.2.

Таблица 6.3. *Полностью обновленная таблица* `Orders`

OrderId	OrderStatus	OrderDate	CustomerId	OrderTotal
1	Отгружен	2020-06-09	100	50.05
2	Отгружен	2020-07-11	101	57.45
3	Отгружен	2020-07-12	102	135.99
4	Отгружен	2020-07-12	100	43.00

В дополнение к таблице `Orders` рассмотрим таблицу `Customers`, показанную в табл. 6.4, которая также полностью загружена в хранилище (т. е. содержит текущее состояние каждой записи о клиенте).

Таблица 6.4. *Полностью обновленная таблица* `Customers`

CustomerId	CustomerName	CustomerCountry
100	Джейн	USA
101	Боб	UK
102	Майлз	UK

Если вы хотите создать эти таблицы в БД для использования в следующих разделах главы, воспользуйтесь приведенными далее SQL-запросами. Обратите внимание, что если вы создали версию таблицы `Orders` в *разделе "Удаление дубликатов записей в таблице"* этой главы, то нужно сначала удалить ее с помощью оператора `DROP`.

```
CREATE TABLE Orders (
  OrderId int,
  OrderStatus varchar(30),
  OrderDate timestamp,
  CustomerId int,
  OrderTotal numeric
);

INSERT INTO Orders VALUES(1,'Отгружен','2020-06-09',100,50.05);
INSERT INTO Orders VALUES(2,'Отгружен','2020-07-11',101,57.45);
INSERT INTO Orders VALUES(3,'Отгружен','2020-07-12',102,135.99);
INSERT INTO Orders VALUES(4,'Отгружен','2020-07-12',100,43.00);

CREATE TABLE Customers
(
  CustomerId int,
  CustomerName varchar(20),
  CustomerCountry varchar(10)
);

INSERT INTO Customers VALUES(100,'Джейн','USA');
INSERT INTO Customers VALUES(101,'Боб','UK');
INSERT INTO Customers VALUES(102,'Майлз','UK');
```

Допустим, нам нужно создать модель данных, отвечающую на следующие вопросы:

- Какой доход был получен от заказов, размещенных в определенной стране в определенном месяце?
- Сколько заказов было размещено в определенный день?

Факты и измерения

Если вы знакомы с *многомерным моделированием* (иногда называемым *моделированием Кимбалла*), то могли заметить, что в описанном примере таблица `Orders` содержит тип данных, который будет смоделирован в *таблице фактов* (fact table), а данные в таблице `Customers` будут смоделированы в *измерении* (dimension). Эти понятия выходят за рамки данной книги, но если вы аналитик данных, я настоятельно рекомендую узнать больше об основах многомерного моделирования. Здесь я буду создавать единую модель данных непосредственно из двух исходных таблиц.

В наших таблицах всего несколько строк, но представьте себе случай, когда обе таблицы содержат миллионы записей. Хотя на вопросы, адресованные к модели данных, можно без особого труда ответить с помощью SQL-запроса, при большом объеме информации время выполнения запроса и объем данных в модели можно сократить, если модель данных в какой-то степени агрегирована.

Если упомянутые вопросы являются единственными двумя требованиями к модели данных, то она должна возвращать две меры:

- общий доход;
- число заказов.

Кроме того, есть два атрибута, по которым модель должна допускать фильтрацию или группировку данных:

- страна заказа;
- дата заказа.

Наконец, уровень детализации модели — по дням, поскольку наименьшая единица времени в потенциальных запросах — один день.

Оптимизация для детализации

Число записей в модели данных зависит от объема данных в исходных таблицах, используемых в модели, количества атрибутов и степени детализации. Всегда выбирайте степень детализации, равную наименьшей

требуемой единице измерения, но не меньше. Если модель должна предоставлять измерения только по месяцам, то детализация по дням не требуется и только увеличит число записей, которые ваша модель должна хранить и запрашивать.

В этой очень упрощенной модели данных я сначала определю структуру модели (таблица SQL), а затем вставлю данные, полученные из объединения обеих таблиц:

```
CREATE TABLE IF NOT EXISTS order_summary_daily (
order_date date,
order_country varchar(10),
total_revenue numeric,
order_count int
);

INSERT INTO order_summary_daily
  (order_date, order_country,
  total_revenue, order_count)
SELECT
  o.OrderDate AS order_date,
  c.CustomerCountry AS order_country,
  SUM(o.OrderTotal) as total_revenue,
  COUNT(o.OrderId) AS order_count
FROM Orders o
INNER JOIN Customers c on
  c.CustomerId = o.CustomerId
GROUP BY o.OrderDate, c.CustomerCountry;
```

Теперь вы можете обратиться к модели, чтобы получить ответ на вопросы, изложенные в требованиях:

```
-- Какой доход был получен от заказов, размещенных в данной стране
в данном месяце?

SELECT
  DATE_PART('month', order_date) as order_month,
  order_country,
  SUM(total_revenue) as order_revenue
FROM order_summary_daily
GROUP BY
  DATE_PART('month', order_date),
  order_country
```

```
ORDER BY
  DATE_PART('month', order_date),
  order_country;
```

Для данных из табл. 6.3 и 6.4 запрос возвращает следующие результаты:

```
order_month | order_country | order_revenue
------------+---------------+--------------
          6 | USA           |         50.05
          7 | UK            |        193.44
          7 | USA           |         43.00
(3 rows)
```

```
-- Сколько заказов было размещено в данный день?
SELECT
  order_date,
  SUM(order_count) as total_orders
FROM order_summary_daily
GROUP BY order_date
ORDER BY order_date;
```

Этот запрос возвращает следующие данные:

```
order_date | total_orders
-----------+-------------
2020-06-09 |            1
2020-07-11 |            1
2020-07-12 |            2
(3 rows)
```

Медленно меняющиеся измерения для полностью обновленных данных

Поскольку данные, полученные в виде полного обновления, перезаписывают изменения существующих данных (например, запись в `Customers`), для отслеживания исторических изменений часто применяется более продвинутая концепция моделирования данных.

Например, в следующем разделе вы будете использовать таблицу `Customers`, которая была загружена инкрементно и содержит обновления `CustomerId 100`. Как вы увидите в табл. 6.6, у этого клиен-

та есть вторая запись, указывающая, что значение CustomerCountry изменилось с USA на UK 2020-06-20. Это означает, что когда клиент разместил заказ с OrderId 4 2020-07-12, на тот момент он не жил в США.

Анализируя историю заказов, аналитик может захотеть распределить заказы по местонахождению клиентов на момент заказа. Как вы увидите в следующем разделе, с инкрементно обновляемыми данными это сделать немного проще. При полностью обновленных данных необходимо вести полную историю таблицы Customers между каждой загрузкой и отслеживать эти изменения самостоятельно.

Соответствующий метод определен в моделировании Кимбалла (многомерном моделировании) и называется *медленно меняющимся измерением* (Slowly Changing Dimension, SCD). При работе с полностью обновленными данными я часто использую SCD типа II, которые добавляют новую запись в таблицу для каждого изменения объекта, включая диапазон дат, когда запись была действительной.

Измерение SCD типа II с записями клиента Джейн будет выглядеть примерно так, как показано в табл. 6.5. Обратите внимание, что срок действия последней записи истекает в очень отдаленном будущем. В некоторых SCD типа II присутствует NULL для записей с неистекшим сроком действия, но, как вы вскоре увидите, отдаленная дата делает запросы к таблице немного менее подверженными ошибкам.

Таблица 6.5. *SCD типа II с данными клиента*

CustomerId	CustomerName	CustomerCountry	ValidFrom	Expired
100	Джейн	USA	2019-05-01 7:01:10	2020-06-20 8:15:34
100	Джейн	UK	2020-06-20 8:15:34	2199-12-31 00:00:00

Вы можете создать и заполнить эту таблицу в своей БД, используя следующие операторы SQL:

```
CREATE TABLE Customers_scd
(
  CustomerId int,
```

```
  CustomerName varchar(20),
  CustomerCountry varchar(10),
  ValidFrom timestamp,
  Expired timestamp
);

INSERT INTO Customers_scd
  VALUES(100,'Джейн','USA','2019-05-01 7:01:10',
    '2020-06-20 8:15:34');
INSERT INTO Customers_scd
  VALUES(100,'Джейн','UK','2020-06-20 8:15:34',
    '2199-12-31 00:00:00');
```

Вы можете объединить SCD с созданной ранее таблицей `Orders`, чтобы получить свойства записи о клиенте на момент заказа. Для этого, помимо объединения по `CustomerId`, вам также необходимо выполнить объединение по диапазону дат в SCD, в котором был размещен заказ. Например, следующий запрос вернет страну, в которой Джейн проживала, согласно записи `Customers_scd`, на момент размещения каждого из ее заказов:

```
SELECT
  o.OrderId,
  o.OrderDate,
  c.CustomerName,
  c.CustomerCountry
FROM Orders o
INNER JOIN Customers_scd c
  ON o.CustomerId = c.CustomerId
    AND o.OrderDate BETWEEN c.ValidFrom AND
c.Expired
ORDER BY o.OrderDate;
```

Результат выполнения запроса:

```
orderid | orderdate           | customer name | customer country
--------+---------------------+---------------+-----------------
      1 | 2020-06-09 00:00:00 | Джейн         | USA
      4 | 2020-07-12 00:00:00 | Джейн         | UK
(2 rows)
```

Хотя подобная логика — это все, что вам нужно для применения SCD в моделировании данных, поддержание SCD в актуальном

состоянии может быть проблемой. В случае с таблицей `Customers` вам нужно будет делать ее снимок после каждого сбора данных и искать любые измененные записи `CustomerId`. Лучший способ решения этой проблемы зависит от того, какое у вас хранилище данных и какие инструменты оркестровки. Если вы заинтересованы в реализации SCD, я советую изучить основы моделирования Кимбалла, что выходит за рамки данной книги. Для более подробного ознакомления с этой темой я рекомендую книгу Ральфа Кимбалла и Марджи Росс "Инструментарий хранилищ данных. Полное руководство по размерному моделированию" (Бомбора, 2023).

Моделирование инкрементно собираемых данных

В *главе 4* было сказано, что инкрементно собираемые данные содержат не только текущее состояние исходных данных, но и исторические записи с момента начала сбора. Например, рассмотрим ту же таблицу `Orders`, что и в предыдущем разделе, но с новой таблицей клиентов с именем `Customers_staging`, которая загружается инкрементно. Как видно из табл. 6.6, появились новые столбцы для значения даты обновления записи `UpdatedDate`, а также новая запись для `CustomerId 100`, указывающая, что страна клиента Джейн (где она живет) изменилась с США на Великобританию 20 июня 2020 г.

Таблица 6.6. *Инкрементно загружаемая таблица* `Customers_staging`

CustomerId	CustomerName	CustomerCountry	LastUpdated
100	Джейн	USA	2019-05-01 7:01:10
101	Боб	UK	2020-01-15 13:05:31
102	Майлз	UK	2020-01-29 9:12:00
100	Джейн	UK	2020-06-20 8:15:34

Вы можете создать и заполнить таблицу `Customers_staging` в своей БД для использования в следующих примерах с помощью операторов SQL:

```
CREATE TABLE Customers_staging (
  CustomerId int,
```

```
  CustomerName varchar(20),
  CustomerCountry varchar(10),
  LastUpdated timestamp
);

INSERT INTO Customers_staging
  VALUES(100,'Джейн','USA','2019-05-01 7:01:10');
INSERT INTO Customers_staging
  VALUES(101,'Боб','UK','2020-01-15 13:05:31');
INSERT INTO Customers_staging
  VALUES(102,'Майлз','UK','2020-01-29 9:12:00');
INSERT INTO Customers_staging
  VALUES(100,'Джейн','UK','2020-06-20 8:15:34');
```

Вспомните вопросы, на которые должна ответить модель из предыдущего раздела и которые мы зададим новой модели:

- Какой доход был получен от заказов, размещенных в определенной стране в определенном месяце?
- Сколько заказов было размещено в определенный день?

В данном случае, прежде чем вы сможете построить свою модель данных, вам нужно решить, как вы хотите обрабатывать изменения в записях в таблице `Customer`. В случае Джейн, к какой стране должны быть отнесены два ее заказа в таблице `Orders`? Должны ли они оба быть отнесены к ее текущей стране проживания (Великобритания) или каждый отдельно к стране, в которой она жила на момент заказа (США и Великобритания соответственно)?

Выбор, который вы делаете, основан на логике бизнес-процессов вашей организации, но реализация в том и другом случае немного отличается. Я начну с привязки к текущей стране. Я сделаю это, создав модель данных, аналогичную описанной в предыдущем разделе, но используя только самую последнюю запись для каждого `CustomerId` в таблице `Customers_staging`. Обратите внимание, что поскольку второй вопрос в требованиях к модели требует детализации по дням, я построю модель на уровне даты:

```
CREATE TABLE order_summary_daily_current
(
  order_date date,
  order_country varchar(10),
  total_revenue numeric,
```

```
  order_count int
);

INSERT INTO order_summary_daily_current
  (order_date, order_country,
  total_revenue, order_count)
WITH customers_current AS
(
  SELECT CustomerId,
    MAX(LastUpdated) AS latest_update
  FROM Customers_staging
  GROUP BY CustomerId
)
SELECT
  o.OrderDate AS order_date,
  cs.CustomerCountry AS order_country,
  SUM(o.OrderTotal) AS total_revenue,
  COUNT(o.OrderId) AS order_count
FROM Orders o
INNER JOIN customers_current cc
  ON cc.CustomerId = o.CustomerId
INNER JOIN Customers_staging cs
  ON cs.CustomerId = cc.CustomerId
    AND cs.LastUpdated = cc.latest_update
GROUP BY o.OrderDate, cs.CustomerCountry;
```

При ответе на вопрос о том, какой доход был получен от заказов, размещенных в определенной стране в определенном месяце, оба заказа Джейн будут отнесены к Великобритании, хотя вы можете ожидать, что ее заказ в июне стоимостью 50.05 будет отнесен к США с учетом ее страны проживания в то время:

```
SELECT
  DATE_PART('month', order_date) AS order_month,
  order_country,
  SUM(total_revenue) AS order_revenue
FROM order_summary_daily_current
GROUP BY
  DATE_PART('month', order_date),
  order_country
ORDER BY
  DATE_PART('month', order_date),
  order_country;
```

```
order_month | order_country | order_revenue
------------+---------------+--------------
          6 | UK            | 50.05
          7 | UK            | 236.44
(2 rows)
```

Если вместо этого вы хотите распределять заказы с учетом страны проживания клиентов на момент заказа, то построение модели потребует изменения логики. Вместо поиска самой последней записи в `Customers_staging` для каждого `CustomerId` в *обобщенном табличном выражении* (Common Table Expression, CTE) я нахожу самую последнюю запись, которая была обновлена в момент оформления или до момента оформления каждого заказа, размещенного каждым клиентом. Другими словами, мне нужна информация о клиенте, которая была действительна на момент размещения заказа. Эта информация хранится в версии их записи `Customer_staging` на момент размещения заказа. Любые более поздние обновления информации о клиенте не происходили до тех пор, пока не был размещен этот конкретный заказ.

Выражение CTE `customer_pit` (слово `pit` — сокращение от point-in-time, *англ.* "момент времени") в следующем примере содержит `MAX(cs.LastUpdated)` для каждой пары `CustomerId/OrderId`. Я использую эту информацию в последнем операторе `SELECT` для заполнения модели данных. Обратите внимание, что в этом запросе я должен выполнить объединение как по `OrderId`, так и по `CustomerId`. Вот окончательный SQL-запрос для модели `order_summary_daily_pit`:

```
CREATE TABLE order_summary_daily_pit
(
  order_date date,
  order_country varchar(10),
  total_revenue numeric,
  order_count int
);

INSERT INTO order_summary_daily_pit
  (order_date, order_country, total_revenue,
order_count)
WITH customer_pit AS
```

```
(
  SELECT
    cs.CustomerId,
    o.OrderId,
    MAX(cs.LastUpdated) AS max_update_date
  FROM Orders o
  INNER JOIN Customers_staging cs
    ON cs.CustomerId = o.CustomerId
      AND cs.LastUpdated <= o.OrderDate
  GROUP BY cs.CustomerId, o.OrderId
)
SELECT
  o.OrderDate AS order_date,
  cs.CustomerCountry AS order_country,
  SUM(o.OrderTotal) AS total_revenue,
  COUNT(o.OrderId) AS order_count
FROM Orders o
INNER JOIN customer_pit cp
  ON cp.CustomerId = o.CustomerId
    AND cp.OrderId = o.OrderId
INNER JOIN Customers_staging cs
  ON cs.CustomerId = cp.CustomerId
    AND cs.LastUpdated = cp.max_update_date
GROUP BY o.OrderDate, cs.CustomerCountry;
```

Когда вы запустите тот же запрос, что и раньше, вы увидите, что доход от первого заказа Джейн в июне 2020 г. ассоциирован с США, а второй заказ в июле 2020 г. ассоциирован с Великобританией, как и ожидалось:

```
SELECT
  DATE_PART('month', order_date) AS order_month,
  order_country,
  SUM(total_revenue) AS order_revenue
FROM order_summary_daily_pit
GROUP BY
  DATE_PART('month', order_date),
  order_country
ORDER BY
  DATE_PART('month', order_date),
  order_country;
```

```
order_month | order_country | order_revenue
------------+---------------+--------------
          6 | USA           | 50.05
          7 | UK            | 236.44
(2 rows)
```

Моделирование данных только для добавления

Данные только для добавления (append-only data, или *данные только для вставки*) — это неизменяемые данные, которые загружаются в хранилище данных. Каждая запись в такой таблице — это какое-то событие, которое никогда не меняется. Примером может служить таблица всех просмотров страниц на веб-сайте. Каждый раз, когда выполняется сбор данных, он добавляет в таблицу новые представления страниц, но никогда не обновляет и не удаляет предыдущие события. Что случилось, то случилось — историю событий нельзя переписать.

Моделирование данных только для добавления аналогично моделированию полностью обновляемых данных. Однако вы можете оптимизировать создание и обновление моделей, построенных на основе таких данных, воспользовавшись тем фактом, что после вставки записей они никогда не изменятся.

Пример таблицы только для добавления под названием `PageViews`, содержащей записи о просмотрах страниц на веб-сайте, приведен в табл. 6.7. Каждая запись в таблице представляет клиента, просматривающего страницу на веб-сайте компании. Новые записи, отражающие просмотры страниц, зарегистрированные с момента последнего сбора данных, добавляются в таблицу каждый раз, когда выполняется задание сбора.

Таблица 6.7. *Пример содержимого таблицы* `PageViews`

CustomerId	ViewTime	UrlPath	utm_medium
100	2020-06-01 12:00:00	/home	соцсеть
100	2020-06-01 12:00:13	/product/2554	NULL
101	2020-06-01 12:01:30	/product/6754	поиск
102	2020-06-02 7:05:00	/home	NULL
101	2020-06-02 12:00:00	/product/2554	соцсеть

Вы можете создать и заполнить таблицу `PageViews` в своей БД для использования в следующих примерах с помощью показанных далее SQL-запросов:

```
CREATE TABLE PageViews (
  CustomerId int,
  ViewTime timestamp,
  UrlPath varchar(250),
  utm_medium varchar(50)
);

INSERT INTO PageViews
  VALUES(100,'2020-06-01 12:00:00','/home','соцсеть');
INSERT INTO PageViews
  VALUES(100,'2020-06-01 12:00:13','/product/2554',NULL);
INSERT INTO PageViews
  VALUES(101,'2020-06-01 12:01:30','/product/6754','поиск');
INSERT INTO PageViews
  VALUES(102,'2020-06-02 7:05:00','/home','NULL');
INSERT INTO PageViews
  VALUES(101,'2020-06-02 12:00:00','/product/2554','соцсеть');
```

Имейте в виду, что реальная таблица с данными о просмотрах страниц будет содержать десятки или более столбцов, в которых хранятся атрибуты просмотренной страницы, URL перехода, версия браузера пользователя и многое другое.

Вспомним парсинг URL-адресов

Таблица `PageViews`, показанная в табл. 6.7, — хороший пример использования методов, описанных в *разделе "Парсинг URL-адресов"* этой главы.

Теперь я дам определение модели данных, которая призвана ответить на следующие вопросы. Я буду использовать таблицу `Customers`, определенную в табл. 6.4 ранее в этой главе, чтобы установить страну, в которой проживает каждый клиент:

- Сколько просмотров страниц регистрируется для каждого URL-адреса на сайте в день?
- Сколько просмотров страниц ежедневно совершают клиенты из каждой страны?

Модель данных детализована на уровне дней. Требуются три атрибута:

- дата просмотра страницы (отметка времени не требуется);
- параметр `UrlPath` просмотра страницы;
- страна, в которой проживает клиент, просматривающий страницу.

Требуется только одна метрика:

- количество просмотров страниц.

Структура модели следующая:

```
CREATE TABLE pageviews_daily (
  view_date date,
  url_path varchar(250),
  customer_country varchar(50),
  view_count int
);
```

Для заполнения модели в первый раз применяется такая же логика, как в *разделе "Моделирование полностью обновляемых данных"* этой главы. Все записи из таблицы `PageViews` включаются в совокупность `pageviews_daily`. В примере кода из листинга 6.4 показан соответствующий SQL-запрос.

Листинг 6.4. Код SQL-запроса pageviews_daily.sql

```
INSERT INTO pageviews_daily
  (view_date, url_path, customer_country, view_count)
SELECT
  CAST(p.ViewTime as Date) AS view_date,
  p.UrlPath AS url_path,
  c.CustomerCountry AS customer_country,
  COUNT(*) AS view_count
FROM PageViews p
LEFT JOIN Customers c ON c.CustomerId = p.CustomerId
GROUP BY
  CAST(p.ViewTime as Date),
  p.UrlPath,
  c.CustomerCountry;
```

Для ответа на один из вопросов, адресованных модели (сколько просмотров страниц клиенты из каждой страны генерируют каждый день?), применяется следующий SQL-запрос:

```
SELECT
  view_date,
  customer_country,
  SUM(view_count)
FROM pageviews_daily
GROUP BY view_date, customer_country
ORDER BY view_date, customer_country;

view_date  | customer_country | sum
-----------+------------------+-----
2020-06-01 | UK               | 1
2020-06-01 | USA              | 2
2020-06-02 | UK               | 2
(3 rows)
```

Теперь подумайте, что нужно сделать, когда произойдет следующий сбор данных в таблицу `PageViews`. При этом будут добавлены новые записи, но все существующие записи остаются нетронутыми. Чтобы обновить модель `pageviews_daily`, у вас есть два варианта:

- очистить таблицу `pageviews_daily` и запустить ту же инструкцию `INSERT`, которую вы использовали для ее заполнения в первый раз. В этом случае вы полностью обновляете модель;
- загружать только новые записи из PageViews в `pageviews_daily`. В этом случае обновление модели происходит постепенно.

Первый вариант наименее сложен и с меньшей вероятностью приведет к логическим ошибкам со стороны аналитика, строящего модель. Если в вашем случае операция `INSERT` выполняется достаточно быстро, я предлагаю выбрать этот путь. Однако будьте осторожны! Хотя полное обновление модели может выполняться достаточно быстро при ее первой разработке, по мере роста наборов данных `PageViews` и `Customers` время выполнения обновления также будет увеличиваться.

Второй вариант немного сложнее, но может привести к сокращению времени выполнения при работе с большими наборами данных. Сложность инкрементного обновления в данном случае заключается в том, что таблица `pageviews_daily` детализирована до уровня дня (дата без метки времени), в то время как новые запи-

си, поступающие в таблицу PageViews, детализированы до полной метки времени.

Почему это проблема? Маловероятно, что вы обновляете таблицу pageviews_daily строго в конце полного дня записей. Другими словами, хотя в pageviews_daily есть данные с датой 2020-06-02, вполне возможно, что новые записи за этот день будут загружены в PageViews при следующем запуске сбора данных.

В табл. 6.8 показан именно такой случай. К предыдущей версии PageViews из табл. 6.7 были добавлены две новые записи. Первый из новых просмотров страниц произошел 2020-06-02, а второй — на следующий день.

Таблица 6.8. *Содержимое таблицы* PageViews *с дополнительными записями*

CustomerId	ViewTime	UrlPath	utm_medium
100	2020-06-01 12:00:00	/home	соцсеть
100	2020-06-01 12:00:13	/product/2554	NULL
101	2020-06-01 12:01:30	/product/6754	поиск
102	2020-06-02 7:05:00	/home	NULL
101	2020-06-02 12:00:00	/product/2554	соцсеть
102	2020-06-02 12:03:42	/home	NULL
101	2020-06-03 12:25:01	/product/567	соцсеть

Прежде чем я попытаюсь инкрементно обновить модель pageviews_daily, взгляните на снимок того, как она выглядит сейчас:

```
SELECT *
FROM pageviews_daily
ORDER BY view_date, url_path, customer_country;
```

```
view_date  | url_path      | customer_country | view_count
-----------+---------------+------------------+-----------
2020-06-01 | /home         | USA              |     1
2020-06-01 | /product/2554 | USA              |     1
2020-06-01 | /product/6754 | UK               |     1
2020-06-02 | /home         | UK               |     1
2020-06-02 | /product/2554 | UK               |     1
(5 rows)
```

Теперь вы можете вставить две новые записи, показанные в табл. 6.8, в свою БД, используя следующие операторы SQL:

```
INSERT INTO PageViews
  VALUES(102,'2020-06-02 12:03:42','/home',NULL);
INSERT INTO PageViews
  VALUES(101,'2020-06-03 12:25:01','/product/567','соцсеть');
```

В качестве первой попытки инкрементного обновления вы можете просто добавить записи из `PageViews` с отметкой времени, превышающей текущее значение `MAX(view_date)` в `pageviews_daily` (2020-06-02). Я попробую это сделать, но вместо вставки в `pageviews_daily` я создам еще одну ее копию с именем `pageviews_daily_2` и использую ее в этом примере. Почему? Потому что, как вы вскоре увидите, предложенная вставка — неправильный подход! SQL будет выглядеть следующим образом:

```
CREATE TABLE pageviews_daily_2 AS
SELECT * FROM pageviews_daily;

INSERT INTO pageviews_daily_2
  (view_date, url_path, customer_country, view_count)
SELECT
  CAST(p.ViewTime as Date) AS view_date,
  p.UrlPath AS url_path,
  c.CustomerCountry AS customer_country,
  COUNT(*) AS view_count
FROM PageViews p
LEFT JOIN Customers c
  ON c.CustomerId = p.CustomerId
WHERE
  p.ViewTime >
  (SELECT MAX(view_date) FROM pageviews_daily_2)
GROUP BY
  CAST(p.ViewTime as Date),
  p.UrlPath,
  c.CustomerCountry;
```

Как вы можете видеть в следующем коде, вы получите несколько повторяющихся записей, потому что все события от 2020-06-02 в полночь и позже включены в обновление. Другими словами, просмотры страниц за 02.06.2020, которые ранее учитывались

в модели, учитываются повторно. Дело в том, что у нас нет полной временной метки, хранящейся в ежедневной детализированной таблице `pageviews_daily` (и копии с именем `pageviews_daily_2`). Если бы эта версия модели использовалась для отчетности или анализа, то число просмотров страниц было бы завышено!

```
SELECT *
FROM pageviews_daily_2
ORDER BY view_date, url_path, customer_country;
```

```
view_date  | url_path      | customer_country | view_count
-----------+---------------+------------------+-----------
2020-06-01 | /home         | USA              | 1
2020-06-01 | /product/2554 | USA              | 1
2020-06-01 | /product/6754 | UK               | 1
2020-06-02 | /home         | UK               | 2
2020-06-02 | /home         | UK               | 1
2020-06-02 | /product/2554 | UK               | 1
2020-06-02 | /product/2554 | UK               | 1
2020-06-03 | /product/567  | UK               | 1
(8 rows)
```

Если вы просуммируете число просмотров по дате, то увидите, что 2020-06-02 было пять просмотров страниц вместо трех, указанных в табл. 6.8. Так произошло потому, что два просмотра страниц за этот день, которые ранее были добавлены в `pageviews_daily_2`, были добавлены снова:

```
SELECT
  view_date,
  SUM(view_count) AS daily_views
FROM pageviews_daily_2
GROUP BY view_date
ORDER BY view_date;
```

```
view_date  | daily_views
-----------+-------------
2020-06-01 |           3
2020-06-02 |           5
2020-06-03 |           1
(3 rows)
```

Другой подход, который применяют многие аналитики, заключается в том, чтобы сохранить полную метку времени последней записи из таблицы `PageViews` и использовать ее в качестве следующей отправной точки для инкрементного обновления. Как и в прошлый раз, я создам новую таблицу (на этот раз с именем `pageviews_daily_3`) для этой попытки, т. к. это снова будет неправильное решение:

```
CREATE TABLE pageviews_daily_3 AS
SELECT * FROM pageviews_daily;

INSERT INTO pageviews_daily_3
  (view_date, url_path, customer_country, view_count)
SELECT
  CAST(p.ViewTime as Date) AS view_date,
  p.UrlPath AS url_path,
  c.CustomerCountry AS customer_country,
  COUNT(*) AS view_count
FROM PageViews p
LEFT JOIN Customers c
  ON c.CustomerId = p.CustomerId
WHERE p.ViewTime > '2020-06-02 12:00:00'
GROUP BY
  CAST(p.ViewTime AS Date),
  p.UrlPath,
  c.CustomerCountry;
```

Если вы посмотрите на новую версию таблицы `pageviews_daily_3`, то заметите нечто странное. Хотя общее число просмотров страниц за 02.06.2020 теперь правильное (3), есть две одинаковые строки (`view_date = 02.06.2020`, `url_path = /home` и `customer_country = UK`):

```
SELECT *
FROM pageviews_daily_3
ORDER BY view_date, url_path, customer_country;
```

```
view_date  | url_path      | customer_country | view_count
-----------+---------------+------------------+-----------
2020-06-01 | /home         | USA              |     1
2020-06-01 | /product/2554 | USA              |     1
2020-06-01 | /product/6754 | UK               |     1
2020-06-02 | /home         | UK               |     1
```

```
2020-06-02 | /home          | UK              |     1
2020-06-02 | /product/2554 | UK              |     1
2020-06-03 | /product/567  | UK              |     1
(7 rows)
```

К счастью, в этом случае ответ на вопрос о том, сколько было просмотров страниц по дням и странам, является правильным. Однако хранить данные, которые нам не нужны, расточительно. Эти две записи можно было бы объединить в одну со значением `view_count`, равным 2. Хотя таблица примера в этом случае небольшая, в действительности такие таблицы нередко содержат много миллиардов записей. Число ненужных дублированных записей увеличивается, что приводит к трате памяти и времени выполнения будущих запросов.

Наилучший подход — предположить, что в течение последнего дня (или недели, месяца и т. д., в зависимости от детализации таблицы) в модель были загружены избыточные данные. Мы выполним следующие действия:

1. Сделаем копию `pageviews_daily` с именем `tmp_page views_daily` со всеми записями от второго до последнего дня, которые она содержит в настоящее время. В нашем случае это означает все данные до 2020-06-01.
2. Вставим все записи из исходной таблицы (`PageViews`) в копию, начиная со следующего дня (2020-06-02).
3. Очистим `pageviews_daily` и загрузим данные из `tmp_pageviews_daily`.
4. Очистим `tmp_pageviews_daily`.

Модифицированный подход

Некоторые аналитики предпочитают придерживаться несколько иного подхода. Вместо того, чтобы очищать `pageviews_daily` на шаге 3, они удаляют `pageviews_daily`, а затем переименовывают `tmp_pageviews_daily` в `pageviews_daily`. Положительным моментом является то, что переменная `pageviews_daily` не пустая между шагами 3 и 4 и может быть запрошена сразу. Недостаток состоит в том, что на некоторых платформах хранилища данных вы потеряете разрешения, установленные для `pageviews_daily`, если они не были скопированы в `tmp_pageviews_daily` на шаге 1. Прежде чем применять этот альтернативный подход, обратитесь к документации по вашей платформе хранилища данных.

Окончательный и правильный SQL-запрос для инкрементного обновления модели выглядит следующим образом:

```
CREATE TABLE tmp_pageviews_daily AS
SELECT *
FROM pageviews_daily
WHERE view_date
  < (SELECT MAX(view_date) FROM pageviews_daily);

INSERT INTO tmp_pageviews_daily
  (view_date, url_path, customer_country, view_count)
SELECT
  CAST(p.ViewTime as Date) AS view_date,
  p.UrlPath AS url_path,
  c.CustomerCountry AS customer_country,
  COUNT(*) AS view_count
FROM PageViews p
LEFT JOIN Customers c
  ON c.CustomerId = p.CustomerId
WHERE p.ViewTime
  > (SELECT MAX(view_date) FROM pageviews_daily)
GROUP BY
  CAST(p.ViewTime as Date),
  p.UrlPath,
  c.CustomerCountry;

TRUNCATE TABLE pageviews_daily;

INSERT INTO pageviews_daily
SELECT * FROM tmp_pageviews_daily;

DROP TABLE tmp_pageviews_daily;
```

Наконец, далее приведен результат правильного инкрементного обновления. Общее число просмотров страниц верное, а данные хранятся максимально эффективно, учитывая требования модели:

```
SELECT *
FROM pageviews_daily
ORDER BY view_date, url_path, customer_country;
```

```
view_date  |    url_path    | customer_country | view_count
-----------+----------------+------------------+-----------
2020-06-01 | /home          | USA              |     1
2020-06-01 | /product/2554  | USA              |     1
2020-06-01 | /product/6754  | UK               |     1
2020-06-02 | /home          | UK               |     2
2020-06-02 | /product/2554  | UK               |     1
2020-06-03 | /product/567   | UK               |     1
(6 rows)
```

Моделирование данных об изменениях

В *главе 4* было сказано, что данные, полученные через CDC, после сбора сохраняются определенным образом в хранилище данных. Например, в табл. 6.9 показано содержимое таблицы `Orders_cdc`, полученной через CDC. Это история трех заказов в исходной системе.

Таблица 6.9. *Содержимое таблицы* `Orders_cdc`

EventType	OrderId	OrderStatus	LastUpdated
insert	1	Размещен	2020-06-01 12:00:00
update	1	Отгружен	2020-06-09 12:00:25
delete	1	Отгружен	2020-06-10 9:05:12
insert	2	Размещен	2020-07-01 11:00:00
update	2	Отгружен	2020-07-09 12:15:12
insert	3	Размещен	2020-07-11 13:10:12

Вы можете создать и заполнить таблицу `Orders_cdc` с помощью следующего SQL-запроса:

```
CREATE TABLE Orders_cdc
(
  EventType varchar(20),
  OrderId int,
  OrderStatus varchar(20),
  LastUpdated timestamp
);
```

```
INSERT INTO Orders_cdc
  VALUES('insert',1,'Размещен','2020-06-01 12:00:00');
INSERT INTO Orders_cdc
  VALUES('update',1,'Отгружен','2020-06-09 12:00:25');
INSERT INTO Orders_cdc
  VALUES('delete',1,' Отгружен ','2020-06-10 9:05:12');
INSERT INTO Orders_cdc
  VALUES('insert',2,' Размещен ','2020-07-01 11:00:00');
INSERT INTO Orders_cdc
  VALUES('update',2,' Отгружен ','2020-07-09 12:15:12');
INSERT INTO Orders_cdc
  VALUES('insert',3,' Размещен ','2020-07-11 13:10:12');
```

Запись заказа 1 впервые была создана при его размещении, но со статусом `Размещен`. Восемь дней спустя запись была обновлена в исходной системе при отправке заказа. Через день запись по какой-то причине была удалена в исходной системе. Заказ 2 прошел аналогичный путь, но никогда не удалялся. Заказ 3 был впервые добавлен при его размещении и никогда не обновлялся. Благодаря CDC мы знаем не только текущее состояние всех заказов, но и их полную историю.

Способ моделирования данных, хранящихся таким образом, зависит от того, на какие вопросы модель данных отвечает. Например, вам может понадобиться отчет о текущем статусе всех заказов для отображения на операционной панели. Возможно, на этой панели необходимо отображать число заказов, находящихся в настоящее время в каждом состоянии. Простая модель может выглядеть примерно так:

```
CREATE TABLE orders_current (
  order_status varchar(20),
  order_count int
);

INSERT INTO orders_current
  (order_status, order_count)
  WITH o_latest AS
  (
    SELECT
       OrderId,
       MAX(LastUpdated) AS max_updated
```

```
    FROM Orders_cdc
    GROUP BY orderid
)
SELECT o.OrderStatus,
  Count(*) as order_count
FROM Orders_cdc o
INNER JOIN o_latest
  ON o_latest.OrderId = o_latest.OrderId
    AND o_latest.max_updated = o.LastUpdated
GROUP BY o.OrderStatus;
```

В рассмотренном примере я использую CTE вместо подзапроса, чтобы найти метку времени `MAX(LastUpdated)` для каждого `OrderId`. Затем я присоединяю полученный CTE к таблице `Orders_cdc`, чтобы получить `OrderStatus` самой последней записи для каждого заказа.

В качестве ответа на первоначальный вопрос вы можете видеть, что два заказа имеют статус `OrderStatus Отправлен`, а один все еще находится в состоянии `Размещен`:

```
SELECT * FROM orders_current;

order_status  | order_count
--------------+-------------
Shipped       |           2
Backordered   |           1
(2 rows)
```

Однако правильный ли это ответ? Напомним, что хотя последним статусом `OrderId 1` в настоящее время было `Отгружен`, запись `Order` была удалена из исходной БД. Хотя это неправильный подход к построению системы, давайте пока будем считать, что когда заказ отменен клиентом, он удаляется из исходной системы. Чтобы принять во внимание факт удаления и проигнорировать удаленные заказы, я внесу небольшую модификацию в обновление модели:

```
TRUNCATE TABLE orders_current;

INSERT INTO orders_current
  (order_status, order_count)
  WITH o_latest AS
```

```
  (
    SELECT
       OrderId,
       MAX(LastUpdated) AS max_updated
    FROM Orders_cdc
    GROUP BY orderid
  )
  SELECT o.OrderStatus,
    Count(*) AS order_count
  FROM Orders_cdc o
  INNER JOIN o_latest
    ON o_latest.OrderId = o_latest.OrderId
      AND o_latest.max_updated = o.LastUpdated
  WHERE o.EventType <> 'delete'
  GROUP BY o.OrderStatus;
```

Как видите, удаленный заказ больше не учитывается:

```
SELECT * FROM orders_current;
```

```
order_status  | order_count
--------------+-------------
Shipped       |           1
Backordered   |           1
(2 rows)
```

Еще одно распространенное использование данных, полученных CDC, — это понимание самих изменений. Допустим, аналитик хочет знать, сколько времени в среднем требуется, чтобы заказы перешли из статуса `Размещен` в `Отгружен`. Я снова буду применять CTE (на этот раз два значения!), чтобы найти первую дату, когда каждый заказ был размещен и отгружен. Затем я вычту вторую дату, чтобы узнать, сколько дней каждый заказ, который был и размещен, и отгружен, находился в состоянии `Размещен`. Обратите внимание, что эта логика намеренно игнорирует `OrderId 3`, который в настоящее время размещен, но еще не отгружен:

```
CREATE TABLE orders_time_to_ship (
  OrderId int,
  backordered_days interval
);
```

```
INSERT INTO orders_time_to_ship
  (OrderId, backordered_days)
WITH o_backordered AS
(
  SELECT
     OrderId,
     MIN(LastUpdated) AS first_backordered
  FROM Orders_cdc
  WHERE OrderStatus = 'Размещен'
  GROUP BY OrderId
),
o_shipped AS
(
  SELECT
     OrderId,
     MIN(LastUpdated) AS first_shipped
  FROM Orders_cdc
  WHERE OrderStatus = 'Отгружен'
  GROUP BY OrderId
)
SELECT b.OrderId,
  first_shipped - first_backordered
    AS backordered_days
  FROM o_backordered b
  INNER JOIN o_shipped s on s.OrderId = b.OrderId;
```

Вы можете увидеть время ожидания каждого заказа, а также использовать функцию `AVG()` для ответа на исходный вопрос:

```
SELECT * FROM orders_time_to_ship;

orderid  | backordered_days
---------+-----------------
       1 | 8 days 00:00:25
       2 | 8 days 01:15:12
(2 rows)

SELECT AVG(backordered_days)
FROM orders_time_to_ship;

avg
-------------------
8 days 00:37:48.5
(1 row)
```

Существует множество других вариантов использования данных, для которых у вас есть полная история изменений, но, как и для моделирования данных, которые были полностью загружены или доступны только для добавления, существуют некоторые общие рекомендации и соображения.

По аналогии с предыдущим разделом, существует потенциальный выигрыш в производительности за счет учета того факта, что данные, принимаемые через CDC, загружаются постепенно, а не полностью обновляются. Однако, как дополнительно отмечалось ранее, бывают случаи, когда прирост производительности не стоит дополнительной сложности, связанной с инкрементным обновлением модели вместо полного обновления. При работе с данными CDC я считаю, что это верно в большинстве случаев. Увеличение сложности работы как с обновлениями, так и с удалениями часто служит убедительным аргументом, чтобы сделать выбор в пользу полного обновления.

Оркестровка конвейеров

В предыдущих главах были описаны основные компоненты конвейеров данных, включая прием данных, преобразование данных и этапы конвейера машинного обучения. В этой главе рассказывается, как связать эти компоненты или этапы в единый слаженный "оркестр".

Оркестровка обеспечивает надлежащее управление зависимостями и выполнение шагов конвейера в правильном порядке.

Когда задача оркестровки конвейеров упоминалась в *главе 2*, я также представил концепцию *платформ оркестровки рабочих процессов*, также называемых *системами управления рабочими процессами* (Workflow Management Systems, WMS), *платформами оркестровки* или *фреймворками оркестровки*). В этой главе я расскажу об Apache Airflow — одном из самых популярных фреймворков такого рода. Хотя основная часть главы посвящена примерам в Airflow, все идеи можно перенести и на другие фреймворки. На самом деле я упомяну несколько альтернатив Airflow позже в этой главе.

Наконец, в последующих разделах этой главы также обсуждаются некоторые более сложные концепции оркестровки конвейеров, включая координацию нескольких конвейеров в вашей инфраструктуре данных.

Направленные ациклические графы

Хотя мы рассматривали направленные ациклические графы (DAG) в *главе 2*, стоит повторить, что они собой представляют. В этой главе рассказывается о том, как они спроектированы и реализованы в Apache Airflow для оркестровки задач в конвейере данных.

Шаги конвейера (задачи) всегда имеют *направленность*, т. е. они начинаются с задачи или нескольких задач и заканчиваются определенной задачей или задачами. Это необходимо, чтобы гарантировать путь выполнения. Другими словами, направленность гарантирует, что новые задачи не будут запущены до тех пор, пока все их зависимые задачи не будут успешно завершены.

Графы конвейеров также должны быть *ациклическими*, т. е. текущая задача не может указывать на ранее выполненную задачу. Другими словами, конвейер не может вернуться назад. Если бы это было возможно, то конвейер мог бы работать бесконечно!

Вспомните пример DAG из *главы 2*, который показан на рис. 7.1. Это DAG, определенный в Apache Airflow.

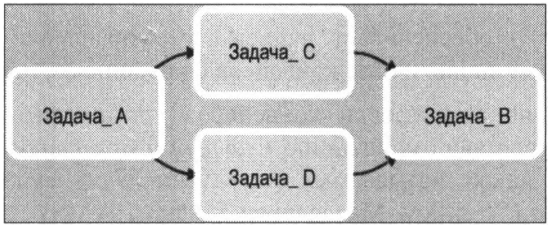

Рис. 7.1. DAG с четырьмя задачами. После завершения задачи *A* запускаются задачи *B* и *C*. Когда они обе завершатся, запускается задача *D*

Задачи в Airflow могут представлять собой что угодно, от выполнения оператора SQL до запуска скрипта Python. Как вы увидите в следующих разделах, Airflow позволяет нам определять, планировать и выполнять задачи в конвейере данных и обеспечивать их запуск в заданном порядке.

Настройка и знакомство с Apache Airflow

Airflow — это проект с открытым исходным кодом, начатый Максимом Бошемином в Airbnb в 2014 г. Он присоединился к программе инкубатора Apache Software Foundation в марте 2016 г. Airflow был создан для решения общей проблемы, с которой регулярно сталкиваются команды инженеров данных: как создавать и контролировать рабочие процессы (в частности, конвейеры данных), включающие в себя несколько задач с взаимными зависимостями.

За шесть лет, прошедших с момента первого выпуска, Airflow стал одним из самых популярных инструментов управления рабочими процессами среди специалистов, работающих с данными. Его простой в использовании веб-интерфейс, расширенные утилиты командной строки, встроенный планировщик и высокий уровень настраиваемости означают, что он хорошо подходит практически для любой инфраструктуры данных. Проект написан на Python, но он может выполнять задачи на любом языке или платформе. На самом деле, хотя Airflow чаще всего применяется для управления конвейерами данных, это действительно универсальная платформа для организации любых зависимых задач.

ПРИМЕЧАНИЕ

Примеры кода и обзор в этой главе относятся к Airflow версии 1.*x*. Airflow 2.0 не за горами и обещает некоторые важные улучшения, такие как новый удобный веб-интерфейс, улучшенный планировщик, полнофункциональный REST API и многое другое. Хотя эта глава ориентирована на использование Airflow 1.*x*, все идеи останутся верными и для Airflow 2.0. Кроме того, представленный здесь программный код предназначен для работы с Airflow 2.0 с небольшими изменениями или без таковых.

Установка и настройка

К счастью, установка Airflow довольно проста. Вам нужно будет использовать менеджер пакетов `pip`, который был представлен в *разделе "Настройка среды Python" главы 4*. После установки и первого запуска Airflow вы познакомитесь с некоторыми его компонентами, такими как БД Airflow, веб-сервер и планировщик. В следующих разделах я расскажу о каждом из них и покажу, как их можно дополнительно настроить.

Airflow в виртуальной среде

Поскольку Airflow построен на Python, вы можете установить Airflow в виртуальную среду Python (virtualenv). Более того, если вы тестируете Airflow или работаете на машине с другими проектами Python, я рекомендую это сделать. Инструкции были приведены в *главе 4* в *разделе "Настройка среды Python"*. Если вы выберете этот метод, обязательно создайте и активируйте рабочую среду, прежде чем продолжить чтение этого раздела.

Вы можете следовать инструкциям по установке из официального руководства по быстрому запуску Airflow. Обычно это занимает не более пяти минут!

После установки Airflow и запуска веб-сервера вы можете ввести адрес **http://localhost:8080** в своем браузере, чтобы получить доступ к веб-интерфейсу Airflow. Если вы хотите узнать больше о различных компонентах Airflow и о том, как их можно настроить, продолжайте чтение этого раздела. Если вы готовы создать свой первый DAG Airflow, то можете сразу перейти к *разделу "Создание DAG Airflow"* этой главы.

Если вас интересует более продвинутая настройка Airflow, то я предлагаю ознакомиться с официальной документацией.

База данных Airflow

Airflow хранит в базе данных все метаданные, связанные с историей выполнения каждой задачи и DAG, а также с вашей конфигурацией Airflow. По умолчанию Airflow взаимодействует с БД SQLite. Когда вы запускали команду `airflow initdb` во время установки, Airflow создал для вас БД SQLite. Для изучения Airflow или даже для небольшого проекта это нормально. Однако для более масштабных задач я предлагаю выбрать БД MySQL или Postgres. К счастью, Airflow "за кадром" использует мощную библиотеку Sql Alchemy и может быть легко перенастроен для работы с другой БД вместо SQLite.

Чтобы изменить БД, которую использует Airflow, вам нужно открыть файл airflow.cfg, который находится по пути, указанному в параметре `AIRFLOW_HOME` во время установки. В примере установки это `~/airflow`. В файле вы увидите строку для конфигурации `sql_alchemy_conn`. Это будет выглядеть примерно так:

```
# Строка подключения SqlAlchemy к базе метаданных.
# SqlAlchemy поддерживает много разных движков БД,
# дополнительная информация доступна на сайте
sql_alchemy_conn = sqlite:////Users/myuser/airflow/airflow.db
```

По умолчанию установлено значение строки подключения для локальной БД SQLite. В следующем примере я создам и настрою БД Postgres и пользователя для Airflow, а затем настрою Airflow

для работы с новой базой данных вместо БД SQLite по умолчанию.

Я предполагаю, что у вас есть работающий сервер Postgres, доступ для запуска psql (интерактивный терминал Postgres) и разрешение на создание баз данных и пользователей в psql. Подойдет любая БД Postgres, но она должна быть доступна с компьютера, на котором работает Airflow. Чтобы узнать больше об установке и настройке сервера Postgres, посетите его официальный сайт. Вы также можете использовать управляемый экземпляр Postgres на такой платформе, как AWS. Это нормально, если машина, на которой установлен Airflow, может получить к нему доступ.

Сначала запустите psql из командной строки или откройте редактор SQL, подключенный к вашему серверу Postgres.

Теперь создайте пользователя для Airflow. Для простоты назовите его `airflow`. Кроме того, установите пароль для пользователя:

```
CREATE USER airflow;
ALTER USER airflow WITH PASSWORD 'pass1';
```

Затем создайте БД для Airflow. Я назову ее `airflowdb`:

```
CREATE DATABASE airflowdb;
```

Наконец, предоставьте новому пользователю все привилегии в новой БД. Airflow потребуется как чтение, так и запись в БД:

```
GRANT ALL PRIVILEGES
  ON DATABASE airflowdb TO airflow;
```

Теперь вы можете вернуться и изменить строку подключения в файле airflow.cfg. Я предполагаю, что ваш сервер Postgres работает на том же компьютере, что и Airflow, но если нет, вам нужно изменить следующую строку, заменив `localhost` полным путем к хосту, на котором работает Postgres. Сохраните airflow.cfg, когда закончите:

```
sql_alchemy_conn = postgresql+psycopg2://
airflow:pass1@localhost:5432/airflowdb
```

Поскольку Airflow потребуется подключиться к БД Postgres через Python, вам также потребуется установить библиотеку psycopg2:

```
$ pip install psycopg2
```

Наконец, вернитесь к командной строке, чтобы повторно инициализировать БД Airflow в Postgres:

```
$ airflow initdb
```

В дальнейшем вы можете найти все метаданные Airflow в БД `airflowdb` на сервере Postgres. Там есть много информации, включая историю задач, которую можно запросить. Вы можете запросить ее непосредственно из БД Postrgres или в веб-интерфейсе Airflow, как описано в следующем разделе. Наличие данных, запрашиваемых с помощью SQL, открывает целый мир возможностей для создания отчетов и анализа. Нет лучшего способа проанализировать производительность ваших конвейеров, и вы можете сделать это с данными, которые Airflow собирает по умолчанию! Использование этих и других данных для измерения и мониторинга производительности ваших конвейеров данных я обсуждаю в *главе 10*.

Веб-сервер и пользовательский интерфейс

Запустив после установки веб-сервер командой `airflow webserver -p 8080`, вы могли украдкой взглянуть, что он собой представляет. Если вы этого еще не сделали, откройте веб-браузер и перейдите по адресу **http://localhost:8080**. Если вы работаете со свежей версией Airflow, то увидите что-то вроде рис. 7.2.

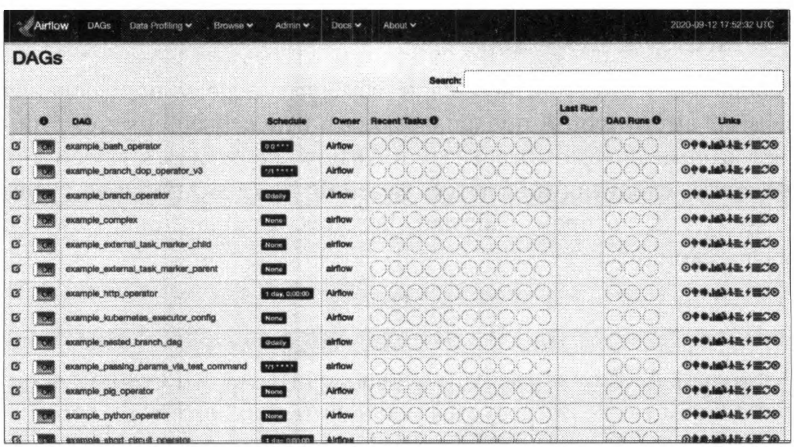

Рис. 7.2. Веб-интерфейс Airflow

На домашней странице веб-интерфейса отображается список DAG. Как видите, Airflow поставляется с некоторыми встроенными примерами DAG. Это отличное место для начала, если вы новичок в Airflow. Когда вы создадите свои собственные DAG, они также появятся там.

На странице есть ряд ссылок и информации для каждого DAG:

- ссылка для просмотра свойств DAG, включая путь к исходному файлу, теги, описание и т. д.;
- переключатель для включения и приостановки DAG. Если этот параметр включен, расписание, указанное в четвертом столбце, определяет время его запуска. При приостановке расписание игнорируется, и DAG может быть запущен только вручную;
- имя DAG, при нажатии на которое открывается страница подробных сведений о DAG, как показано на рис. 7.3;
- расписание, по которому работает DAG, когда он не приостановлен. Расписание показано в формате crontab и определено в исходном файле DAG.

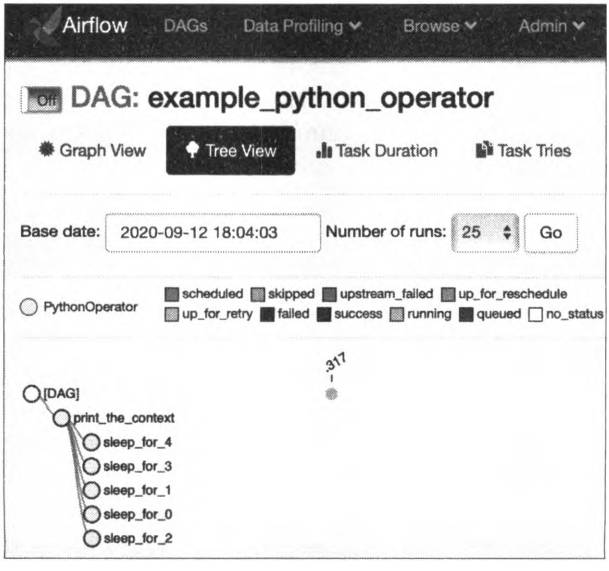

Рис. 7.3. Древовидное представление DAG Airflow

- владелец DAG. Обычно это Airflow, но в более сложных системах у вас может быть несколько владельцев на выбор;
- недавние задачи — сводка последнего запуска DAG;
- временная метка последнего запуска DAG;
- сводка предыдущих запусков DAG;
- набор ссылок на различные конфигурации и информацию DAG. Вы также увидите эти ссылки, если нажмете на имя DAG.

Нажав на имя DAG, вы перейдете к древовидному представлению DAG на странице сведений о графе, как показано на рис. 7.3. Это DAG `example_python_operator`, который поставляется с Airflow. В DAG есть пять задач, каждая из которых является оператором Python `PythonOperators` (об операторах вы узнаете позже в этом разделе). После успешного завершения задачи `print_the_context` запускаются пять задач. Когда они будут выполнены, выполнение DAG тоже будет завершено.

Вы также можете нажать кнопку **Graph View** (Просмотр графа) в верхней части страницы, чтобы увидеть, как DAG выглядит в виде схемы. Я считаю это представление наиболее полезным. На рис. 7.4 показано, как выглядит этот конкретный граф.

В более сложных DAG с большим числом задач графическое представление трудно целиком разглядеть на экране. Однако вы можете увеличивать и уменьшать масштаб и прокручивать график с помощью мыши.

На экране есть ряд других опций, названия которых говорят сами за себя. Но я хотел бы сосредоточиться на двух из них: **Code** (Код) и **Trigger DAG** (Запуск DAG).

Когда вы нажмете кнопку **Code**, то, конечно же, увидите код DAG. Прежде всего, вы заметите, что DAG определен в скрипте Python. В данном случае файл называется example_python_operator.py. Позже в этой главе вы узнаете больше о структуре исходного файла DAG. На данный момент важно знать, что он содержит конфигурацию DAG, включая его расписание, определение каждой задачи и зависимости между каждой задачей.

Кнопка **Trigger DAG** позволяет запускать DAG по требованию. Хотя Airflow создан для запуска DAG по расписанию, эта кноп-

ка — самый простой способ запустить DAG немедленно во время разработки, во время тестирования и для неотложных потребностей в производстве.

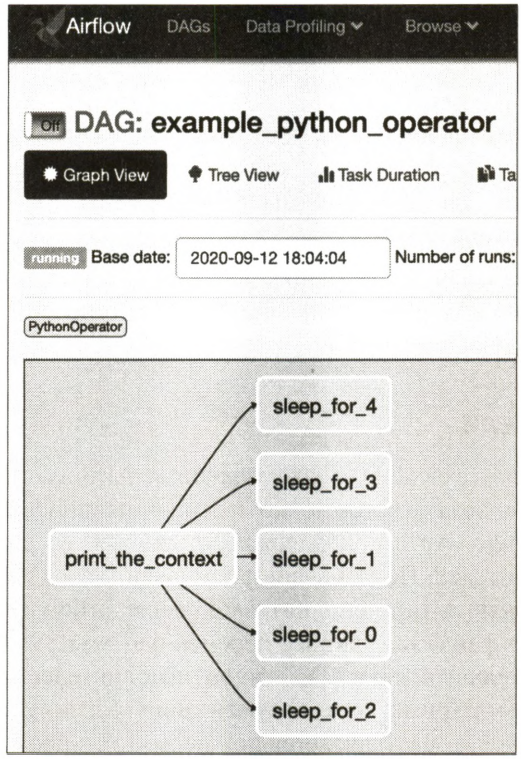

Рис. 7.4. Визуальное представление DAG Airflow

Помимо управления DAG, веб-интерфейс оснащен рядом других полезных функций. Если вы нажмете кнопку **Data Profiling** (Профилирование данных) на верхней панели навигации, то увидите опции выбора **Ad Hoc Query** (Прямой запрос), **Charts** (Диаграммы) и **Known Events** (Известные события). Здесь вы можете запросить информацию из БД Airflow, если не хотите подключаться к ней напрямую из другого инструмента.

В разделе **Browse** (Обзор) вы можете найти историю выполнения DAG и других файлов журналов, а в разделе **Admin** (Администратор) доступны различные параметры конфигурации. Вы може-

те узнать больше о дополнительных параметрах конфигурации в официальной документации Airflow.

Планировщик

Планировщик Airflow — это служба, которую вы запустили при выполнении команды `airflow scheduler` ранее в этой главе. Во время работы планировщик постоянно отслеживает DAG и задачи и запускает те из них, которые были запланированы для запуска или для которых были выполнены их зависимости (в случае задач в DAG).

Для запуска задач планировщик использует исполнитель, определенный в разделе `[core]` файла airflow.cfg. Об исполнителях говорится в следующем разделе.

Исполнители

Исполнитель (executor) — это механизм Airflow, применяемый для запуска задач, которые по мнению планировщика готовы к выполнению. Airflow поддерживает множество различных типов исполнителей. По умолчанию назначен `SequentialExecutor`. Вы можете изменить тип исполнителя в файле airflow.cfg. В разделе `[core]` этого файла вы увидите переменную `executor`, для которой можно установить любой из типов исполнителей, перечисленных в этом разделе и в документации Airflow. Как видите, `SequentialExecutor` настраивается при первой установке Airflow:

```
[core]
........
# Класс исполнителя, предназначенного для запуска.
Choices include
# SequentialExecutor, LocalExecutor,
CeleryExecutor, DaskExecutor, KubernetesExecutor
executor = SequentialExecutor
```

Хотя `SequentialExecutor` задан по умолчанию, он не предназначен для производственных сценариев, поскольку может выполнять только одну задачу за раз. Это приемлемо для тестирования простых DAG, но не более того. Однако это единственный исполнитель, совместимый с БД SQLite, поэтому, если вы не настроили

другую базу данных с помощью Airflow, то `SequentialExecutor` — ваш единственный вариант.

Если вы планируете более масштабное производственное применение Airflow, я предлагаю выбрать другой исполнитель, такой как `CeleryExecutor`, `DaskExecutor` или `KubernetesExecutor`. Ваш выбор отчасти должен зависеть от того, какая инфраструктура вам наиболее удобна. Например, чтобы включить `CeleryExecutor`, вам нужно настроить брокер Celery, используя RabbitMQ, Amazon SQL или Redis.

Настройка инфраструктуры, необходимой для каждого исполнителя, выходит за рамки этой книги, но примеры в этом разделе будут выполняться даже на `SequentialExecutor`. Вы можете узнать больше об исполнителях Airflow в документации.

Операторы

Напомним, что каждый из узлов в DAG является задачей. В Airflow каждая задача реализована через *операторы*. Операторы — это механизм, который фактически выполняет сценарии, команды и другие операции. Существует довольно много операторов. Вот наиболее распространенные:

- `BashOperator`;
- `PythonOperator`;
- `SimpleHttpOperator`;
- `EmailOperator`;
- `SlackAPIOperator`;
- `MySqlOperator`, `PostgresOperator` и другие специфичные для БД операторы для выполнения команд SQL;
- `Sensor`.

Как вы узнаете из следующего раздела, операторы создаются и назначаются каждой задаче в DAG.

Создание DAG Airflow

Теперь, когда вы знаете, как работает Airflow, пришло время создать DAG! Хотя Airflow поставляется с набором примеров DAG, я собираюсь продолжить примеры, приведенные ранее в этой книге, и создать DAG, который выполняет шаги примера процесса ELT. В частности, он будет извлекать данные из БД, загружать их в хранилище, а затем преобразовывать данные в модель данных.

Простой DAG

Прежде чем я создам пример DAG для ELT, важно понять, как DAG определяются в Airflow. Каждый DAG определяется в сценарии Python, где его структура и зависимости задач записываются в коде Python. Листинг 7.1 — это определение простого DAG с тремя задачами. Такой код называется *файлом определения* DAG. Каждая задача определяется как `BashOperator`, при этом первая и третья задачи распечатывают заданный текст, а вторая создает паузу на три секунды. Хотя этот код не делает ничего особенно полезного, он полностью функционален и содержит определения DAG, которые вы напишете позже.

Листинг 7.1. Файл определения DAG simple_dag.py

```
from datetime import timedelta
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.utils.dates import days_ago

dag = DAG(
    'simple_dag',
    description='A simple DAG',
    schedule_interval=timedelta(days=1),
    start_date = days_ago(1),
)

t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag,
)
```

```
t2 = BashOperator(
    task_id='sleep',
    depends_on_past=False,
    bash_command='sleep 3',
    dag=dag,
)

t3 = BashOperator(
    task_id='print_end',
    depends_on_past=False,
    bash_command='echo \'end\'',
    dag=dag,
)

t1 >> t2
t2 >> t3
```

Прежде чем вы двинетесь дальше и запустите DAG, я хотел бы указать на ключевые особенности файла определения DAG. Сначала, как и в любом скрипте Python, импортируются необходимые модули. Затем определяется сам DAG, и ему присваиваются некоторые свойства, такие как имя (`simple_dag`), расписание, дата начала и т. д. На самом деле, есть много других свойств, не упомянутых в этом простом примере. Если они вам понадобятся, то вы можете найти их позже в этой главе или в официальной документации Airflow.

Далее я определяю три задачи в DAG. Все они относятся к типу `BashOperator`, т. е. при выполнении они запускают команду `bash`. Каждой задаче также присваивается несколько свойств, в том числе буквенно-цифровой идентификатор, называемый `task_id`, а также команда `bash`, которая запускается при выполнении задачи. Как вы увидите позже, каждый тип оператора имеет свои собственные настраиваемые свойства, аналогично тому, как у `BashOperator` есть `bash_command`.

Последние две строки определения DAG устанавливают зависимости между задачами. Их можно прочитать так: когда задача `t1` завершается, запускается `t2`. Когда `t2` завершается, запускается `t3`. При просмотре DAG в веб-интерфейсе Airflow вы увидите, что эта последовательность отражена как в виде дерева, так и в виде графа.

Чтобы запустить DAG, вам нужно сохранить файл определения в том месте, где Airflow ищет DAG. Вы можете найти это местоположение (или изменить его) в файле airflow.cfg:

```
dags_folder = /Users/myuser/airflow/dags
```

Сохраните определение DAG в файле с именем simple_dag.py и поместите его в папку dags_folder. Если у вас уже запущен веб-интерфейс Airflow и планировщик, обновите веб-интерфейс Airflow, и вы должны увидеть в списке DAG с именем simple_dag. Если его не видно, подождите несколько секунд и повторите попытку или остановите и перезапустите веб-службу.

Затем нажмите на имя DAG, чтобы просмотреть его более подробно. Вы сможете увидеть графовое и древовидное представление DAG, а также код, который вы только что написали. Готовы к запуску? Либо на этом экране, либо на домашней странице установите переключатель режима DAG в положение **On** (Включен), как показано на рис. 7.5.

Рис. 7.5. Включение DAG

Напомню, что в коде для свойства `schedule_interval` DAG задано значение `timedelta(days=1)`. Это означает, что DAG запускается один раз в сутки в полночь по UTC. Вы увидите, что это расписание отображается как на домашней странице Airflow рядом с DAG, так и на странице сведений о DAG. Также обратите внимание, что для свойства `start_date` DAG установлено значение `days_ago(1)`. Это означает, что первый запуск DAG устанавливается за один день до текущего дня. Если для DAG установлено значение **On**, то первый запланированный запуск — 0:00:00 UTC дня, предшествующего текущему дню, и, таким образом, он начнет выполняться, как только исполнитель станет доступным.

Даты запуска и остановки DAG

В этом примере я установил `start_date` для DAG на один день раньше, чтобы, как только я включу его, запуск был запланирован и выполнен. Иногда целесообразно жестко запрограммировать дату в будущем, ска-

жем, день, когда вы хотите, чтобы DAG впервые запустился в производственном развертывании Airflow. Вы также можете установить для DAG дату остановки `end_date`, чего я не сделал. Если оба этих параметра не указаны, то DAG будет запускаться каждый день в течение неограниченного срока.

Вы можете проверить статус запуска DAG на странице сведений о DAG или перейдя во вкладку **Browse → DAG Runs** (Запуски DAG) в верхнем меню. Там вы должны увидеть визуальный статус запуска DAG, а также каждой задачи в DAG. На рис. 7.6 показан запуск примера `simple_dag`, в котором все задачи выполнены успешно. Окончательный статус DAG отмечен как **success** (успешно) в левой части экрана.

Если вы хотите запустить DAG вручную, нажмите кнопку **Trigger DAG** на странице сведений о DAG.

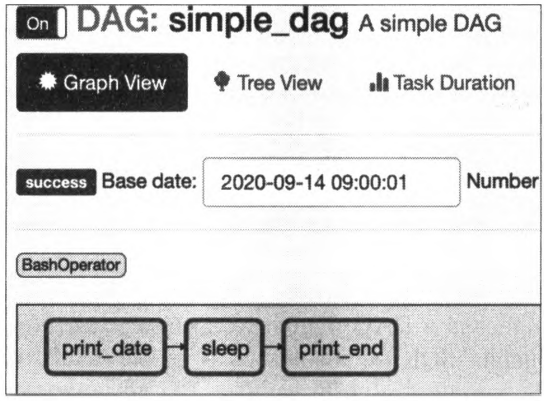

Рис. 7.6. Визуальное представление DAG Airflow

Конвейер ELT и DAG

Теперь, когда мы узнали, как создать простой DAG, вы можете разработать свой DAG для этапов извлечения, загрузки и преобразования конвейера данных. Этот DAG состоит из пяти задач.

Первые две задачи используют `BashOperators` для выполнения двух разных сценариев Python, каждый из которых извлекает данные из таблицы БД Postgres и отправляет результаты в виде CSV-файла в корзину S3. Хотя я не буду воссоздавать здесь логику

сценариев, вы можете найти ее в *разделе "Извлечение данных из БД PostgreSQL" главы 4*. Фактически вы можете выбрать любой пример извлечения из этой главы, если хотите выполнить извлечение из БД MySQL, REST API или MongoDB.

После завершения всех упомянутых задач выполняется соответствующая задача по загрузке данных из корзины S3 в хранилище данных. Опять же, каждая задача использует `BashOperator` для выполнения скрипта Python, содержащего логику для загрузки файла CSV. Вы можете найти пример кода в *главе 5* в *разделе "Загрузка данных в хранилище данных Snowflake"* или *"Загрузка данных в хранилище Redshift"*, в зависимости от вашей платформы.

Варианты запуска кода Python

В этом примере для выполнения сценариев Python я использую `BashOperator` вместо `PythonOperator`. Почему? Чтобы выполнить код Python с помощью `PythonOperator`, код должен быть либо записан в файле определения DAG, либо импортирован в него. Хотя это можно было сделать в коде примера, я хотел бы сохранить большее разделение между моей оркестровкой и логикой процессов, которые она выполняет. Во-первых, таким образом я избегаю потенциальных проблем несовместимости версий библиотек Python между Airflow и любым моим собственным кодом, который я хочу выполнить. В целом мне проще поддерживать логику в инфраструктуре данных, разделяя проекты (и репозитории Git). Логика оркестровки и конвейера ничем не отличается.

Последняя задача в DAG использует `PostgresOperator` для выполнения сценария SQL (хранящегося в файле *.sql) в хранилище данных для создания модели данных. Вы помните этот сценарий из *главы 6*. Вместе эти пять задач образуют простой конвейер, построенный по шаблону ELT, впервые представленному в *главе 3*.

На рис. 7.7 показано графовое представление DAG.

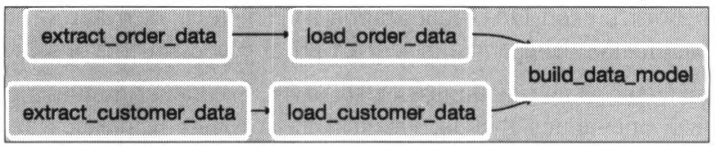

Рис. 7.7. Графовое представление примера DAG ELT

В листинге 7.2 показано определение DAG. Найдите минутку, чтобы изучить его, хотя я также предоставлю описание. Вы можете сохранить его в папке Airflow dags, но пока не включайте его.

Листинг 7.2. Пример определения DAG в файле elt_pipeline_sample.py

```
from datetime import timedelta
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.postgres_operator import PostgresOperator
from airflow.utils.dates import days_ago

dag = DAG(
    'elt_pipeline_sample',
    description='A sample ELT pipeline',
    schedule_interval=timedelta(days=1),
    start_date = days_ago(1),
)

extract_orders_task = BashOperator(
    task_id='extract_order_data',
    bash_command='python /p/extract_orders.py',
    dag=dag,
)

extract_customers_task = BashOperator(
    task_id='extract_customer_data',
    bash_command='python /p/extract_customers.py',
    dag=dag,
)

load_orders_task = BashOperator(
    task_id='load_order_data',
    bash_command='python /p/load_orders.py',
    dag=dag,
)

load_customers_task = BashOperator(
    task_id='load_customer_data',
```

```
    bash_command='python /p/load_customers.py',
    dag=dag,
)

revenue_model_task = PostgresOperator(
    task_id='build_data_model',
    postgres_conn_id='redshift_dw',
    sql='/sql/order_revenue_model.sql',
    dag=dag,
)

extract_orders_task >> load_orders_task
extract_customers_task >> load_customers_task
load_orders_task >> revenue_model_task
load_customers_task >> revenue_model_task
```

Из листинга 7.1 вы узнали, как импортировать некоторые необходимые пакеты Python и создавать объект DAG. На этот раз нужно импортировать еще один пакет, чтобы использовать `PostgresOperator` в последней задаче DAG. Этот DAG, как и предыдущий пример, планируется запускать один раз в сутки в полночь, начиная с предыдущего дня.

Последняя задача посредством `PostgresOperator` выполняет сценарий SQL, хранящийся в каталоге на том же компьютере, что и Airflow в хранилище данных. Содержимое сценария SQL похоже на преобразование модели данных из *главы 6*. Например, учитывая, что DAG извлекает и загружает таблицы `Orders` и `Customers`, я буду использовать следующий пример из *главы 6*. Вы можете, конечно, взять любой SQL-запрос для сопоставления данных, с которыми вы работаете.

```
CREATE TABLE IF NOT EXISTS order_summary_daily (
order_date date,
order_country varchar(10),
total_revenue numeric,
order_count int
);

INSERT INTO order_summary_daily
  (order_date, order_country,
  total_revenue, order_count)
```

```
SELECT
  o.OrderDate AS order_date,
  c.CustomerCountry AS order_country,
  SUM(o.OrderTotal) AS total_revenue,
  COUNT(o.OrderId) AS order_count
FROM Orders o
INNER JOIN Customers c
  ON c.CustomerId = o.CustomerId
GROUP BY
  o.OrderDate, c.CustomerCountry;
```

Прежде чем включить DAG, нужно сделать еще один шаг — установить соединение, которое будет использоваться для `PostgresOperator`. Как видите, в определении DAG есть параметр `postgres_conn_id` со значением `redshift_dw`. Вам нужно будет определить соединение `redshift_dw` в веб-интерфейсе Airflow, чтобы `PostgresOperator` мог выполнить скрипт.

Для этого выполните следующие действия:

1. Откройте веб-интерфейс Airflow и выберите разделы **Admin** → **Connections** (Администратор → Подключения) на верхней панели навигации.
2. Нажмите на вкладку **Create** (Создать).
3. Установите для параметра **Conn ID** значение redshift_dw (или любой другой идентификатор, который вы хотите использовать в файле определения DAG).
4. Выберите Postgres в качестве типа соединения.
5. Задайте информацию о соединении для вашей БД.
6. Нажмите кнопку **Save** (Сохранить).

Amazon Redshift совместим с соединениями Postgres, поэтому я выбрал этот тип соединения. Вам также доступны соединения для Snowflake и десятков других баз данных и платформ, таких как Spark.

Теперь все готово, чтобы включить DAG. Вы можете вернуться на домашнюю страницу или открыть страницу сведений о DAG и щелкнуть переключатель, чтобы установить для DAG значение **On**. Поскольку расписание DAG начинается ежедневно в полночь предыдущего дня, выполнение будет начато немедленно. Вы мо-

жете проверить статус запуска DAG на странице сведений о DAG или перейдя к вкладкам **Browse → DAG Runs** (Обзор → Запуски DAG) в верхнем меню. Как всегда, вы можете инициировать однократный запуск DAG с помощью кнопки **Trigger DAG** на странице сведений о DAG.

Хотя этот пример немного упрощен, он содержит этапы конвейера ELT. В более сложном конвейере вы найдете гораздо больше задач. В дополнение к большему количеству извлечений и загрузок данных, вероятно, будет много моделей данных, причем некоторые из них могут зависеть друг от друга. Airflow позволяет легко обеспечить их выполнение в правильном порядке. В большинстве производственных развертываний Airflow вы найдете множество DAG для конвейеров, которые могут иметь определенную зависимость друг от друга или от какой-либо внешней системы или процесса. Обратитесь к *разделу "Расширенные конфигурации оркестровки"* этой главы, где приведены некоторые советы по решению таких задач.

Дополнительные задачи конвейера

В дополнение к функциональным задачам в примере конвейера ELT, рассмотренном в предыдущем разделе, для конвейеров производственного уровня требуются другие задачи, такие как отправка уведомлений в Slack-канал, когда конвейер завершается или выходит из строя, выполнение проверки данных в различных точках конвейера, и т. д. К счастью, со всеми этими задачами может справиться DAG Airflow.

Оповещения и уведомления

Хотя веб-интерфейс Airflow — отличное место для просмотра статуса выполнения DAG, часто лучше получать электронное письмо, когда DAG терпит неудачу (или даже когда сработал успешно). Существует несколько вариантов отправки уведомлений. Например, если вы хотите получать электронное письмо при сбое DAG, то можете добавить следующие параметры при создании экземпляра объекта DAG в файле определения. Вы также можете добавить их в задачи вместо DAG, если хотите получать уведомления только для определенных задач:

```
'email': ['me@example.com'],
'email_on_failure': True,
```

Прежде чем Airflow сможет отправить вам электронное письмо, необходимо предоставить сведения о вашем SMTP-сервере в разделе `[smtp]` файла airflow.cfg.

Вы также можете задать `EmailOperator` в задаче для отправки электронной почты в любой точке DAG:

```
email_task = EmailOperator(
            task_id='send_email',
            to="me@example.com",
            subject="Airflow Test Email",
            html_content='Пробный текст письма',
)
```

В дополнение к `EmailOperator` существуют как официальные, так и поддерживаемые сообществом операторы для отправки сообщений в Slack, Microsoft Teams и другие платформы. Конечно, вы всегда можете создать свой собственный скрипт Python для отправки сообщения на выбранную вами платформу и выполнить его с помощью `BashOperator`.

Проверка данных

В *главе 8* более подробно обсуждается проверка данных и конвейеры тестирования, но добавление задачи в DAG Airflow для запуска проверки данных — это отдельная и настоятельно рекомендуемая операция. Как вы узнаете из этой главы, проверка данных может быть реализована в сценарии SQL или Python или путем вызова какого-либо другого внешнего приложения. Теперь вы знаете, что Airflow может справиться со всеми подобными задачами!

Расширенные конфигурации оркестровки

В предыдущем разделе был представлен пример простого DAG, который запускает полный сквозной конвейер данных, соответствующий шаблону ELT. Теперь рассмотрим некоторые проблемы, с которыми вы можете столкнуться при построении более

сложных конвейеров или при необходимости координировать несколько конвейеров с общими зависимостями или разными расписаниями.

Связанные и несвязанные задачи конвейера

Хотя из приведенных ранее примеров может показаться, что все шаги (задачи) в конвейере данных четко связаны друг с другом, это не всегда так. Возьмем потоковую передачу данных. Допустим, Kafka используется для потоковой передачи данных в корзину S3, откуда они непрерывно загружаются в хранилище данных Snowflake с помощью Snowpipe (см. *главы 4* и *5*).

В этом случае данные непрерывно поступают в хранилище, но этап преобразования данных по-прежнему будет выполняться с определенным интервалом, например каждые 30 минут. В отличие от DAG в листинге 7.2, конкретные запуски сбора данных не являются прямыми зависимостями от задачи по преобразованию данных в модель. В такой ситуации говорят, что задачи *не связаны*, в отличие от *связанных* задач в DAG.

Учитывая эту реальность, инженеры по обработке данных должны тщательно подходить к организации конвейеров. Хотя жестких правил нет, чтобы управлять несвязанными задачами, необходимо принимать последовательные и надежные решения во всех конвейерах. В примере потокового приема данных и запланированного шага преобразования логика преобразования должна учитывать, что данные из двух различных источников (например, таблицы `Orders` и `Customers`) могут находиться в несколько разных состояниях обновления. Логика преобразования должна учитывать случаи, когда, например, имеется запись `Order` без соответствующей записи `Customer`.

Когда следует разделять DAG

Ключевой момент принятия решения при проектировании конвейеров — определение задач, которые взаимодействуют в DAG. Хотя можно создать DAG со всеми задачами извлечения, загрузки, преобразования, проверки и оповещения в вашей инфраструктуре данных, он довольно быстро станет слишком сложным.

Три фактора влияют на определение того, когда задачи должны быть разбиты на несколько DAG, а когда они должны оставаться в одном месте:

- Когда задачи должны выполняться по разным расписаниям, разбейте их на несколько DAG.

  Если у вас одни задачи запускаются только один раз в сутки, а другие запускаются каждые 30 минут, то, вероятно, следует разделить их на два DAG. В противном случае вы потратите время и ресурсы на выполнение некоторых задач по 47 лишних раз в день! В ситуациях, когда затраты на вычисления определяются фактическим использованием, это очень важно.

- Когда конвейер действительно независим, держите его отдельно.

  Если задачи в конвейере связаны только друг с другом, держите их в одном DAG. Возвращаясь к примеру из листинга 7.2, если сбор данных из таблиц `Orders` и `Customer` используется только моделью данных в этом DAG и никакие другие задачи не зависят от модели данных, то имеет смысл оставить такой DAG обособленным.

- Когда DAG становится слишком сложным, задумайтесь, можно ли подвергнуть его логическому разбиению.

  Хотя это субъективный критерий, но если вы видите пугающее графовое представление DAG с сотнями задач и паутиной из стрелок зависимостей, значит, пришло время подумать о том, как его разбить. В противном случае вам может быть трудно поддерживать такой граф в будущем.

Работа с несколькими DAG, которые могут иметь общие зависимости (например, прием данных), может показаться лишней головной болью, но часто без этого не обойтись. В следующем разделе я расскажу, как реализовать зависимости между DAG в Airflow.

Координация нескольких DAG с сенсорами

Отвечая на потребность в общих зависимостях между DAG, задачи Airflow могут реализовывать специальный тип оператора — так называемый `Sensor`. Он предназначен для проверки состояния

некоторой внешней задачи или процесса, а затем продолжения выполнения нижестоящих зависимостей в своем DAG при соблюдении критериев проверки.

Если вам необходимо координировать два разных DAG Airflow, вы можете с помощью оператора `ExternalTaskSensor` проверить состояние задачи в другом DAG или состояние другого DAG в полном объеме. В листинге 7.3 определяется DAG с двумя задачами. Первая использует `ExternalTaskSensor` для проверки состояния DAG `elt_pipe line_sample` из предыдущего раздела этой главы. Когда этот DAG завершается, `Sensor` возвращает показание "успешно", и выполняется следующая задача (`task1`).

Листинг 7.3. Использование сенсора, файл sensor_test.py

```
from datetime import datetime
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.sensors.external_task_sensor import ExternalTaskSensor
from datetime import timedelta
from airflow.utils.dates import days_ago

dag = DAG(
        'sensor_test',
        description='DAG with a sensor',
        schedule_interval=timedelta(days=1),
        start_date = days_ago(1))

sensor1 = ExternalTaskSensor(
            task_id='dag_sensor',
            external_dag_id = 'elt_pipeline_sample',
            external_task_id = None,
            dag=dag,
            mode = 'reschedule',
            timeout = 2500)

task1 = DummyOperator(
            task_id='dummy_task',
            retries=1,
            dag=dag)

sensor1 >> task1
```

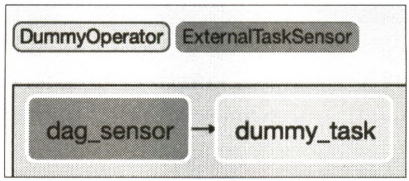

Рис. 7.8. Графовое представление примера ELT DAG

На рис. 7.8 показано графовое представление DAG.

В начале работы этот DAG сначала запускает задачу `dag_sensor`. Обратите внимание на его свойства:

- для `external_dag_id` задан идентификатор DAG, которую будет отслеживать датчик. В данном случае это группа DAG `elt_pipeline_sample`;
- в этом примере для свойства `external_task_id` установлено значение `None`, означающее, что датчик ожидает успешного завершения всего DAG `elt_pipeline_sample`. Если бы вместо этого вы указали конкретный идентификатор задачи в DAG `elt_pipeline_sample`, то сразу по окончании задачи с этим идентификатором `sensor1` завершился бы и запустил `dummy_task`.
- свойство `mode` установлено в `reschedule`. По умолчанию датчики работают в режиме `poke`, при котором сенсор блокирует рабочий слот, "зондируя" внешнюю задачу. В зависимости от того, какой исполнитель задействован и сколько задач выполняется, это не всегда идеально. В режиме `reschedule` рабочий слот освобождается путем перепланирования задачи и, таким образом, открытия рабочего слота до тех пор, пока он не будет настроен для повторного запуска;
- для параметра `timeout` установлено число секунд, в течение которых `ExternalTaskSensor` будет продолжать проверять свою внешнюю зависимость до истечения времени ожидания. Здесь рекомендуется задавать разумный тайм-аут; в противном случае DAG будет работать вечно.

Следует иметь в виду, что DAG запускаются по определенному расписанию, поэтому сенсору необходимо проверять факт запуска определенного DAG. По умолчанию `ExternalTaskSensor` проверяет запуск `external_dag_id` с текущим расписанием DAG, к кото-

рому он принадлежит. Поскольку DAG `elt_pipeline_sample` и `sensor_test` запускаются один раз в сутки в полночь, можно оставить значение по умолчанию. Однако если два DAG работают по разным расписаниям, то лучше указать, какой запуск `elt_pipeline_sample` должен проверять сенсор. Вы можете сделать это, используя либо параметр `execute_delta`, либо параметр `execute_date_fn` сенсора `ExternalTaskSensor`. Параметр `execute_date_fn` определяет конкретную дату и время запуска DAG, и я считаю его менее полезным, чем `execute_delta`.

Параметр `execute_delta` пригоден для ретроспективного анализа конкретного запуска DAG. Например, чтобы просмотреть самый последний запуск DAG, запланированный на каждые 30 минут, вы должны создать задачу, которая определяется следующим образом:

```
sen1 = ExternalTaskSensor(
          task_id='dag_sensor',
          external_dag_id = 'elt_pipeline_sample',
          external_task_id = None,
          dag=dag,
          mode = 'reschedule',
          timeout = 2500,
          execution_delta=timedelta(minutes=30))
```

Управляемые варианты развертывания Airflow

Хотя развертывание простого экземпляра Airflow не вызывает затруднений, в производственных масштабах это становится гораздо более сложной задачей. Работа с более сложными исполнителями для большего параллелизма задач, поддержание вашего экземпляра в актуальном состоянии и масштабирование базовых ресурсов — это задачи, на которые не у каждого инженера данных есть время.

Как и для многих других инструментов с открытым исходным кодом, для Airflow существует несколько полностью управляемых внешних решений. Два наиболее известных — Cloud Composer в Google Cloud и Astronomer. Хотя ваша ежемесячная плата

намного превысит стоимость запуска Airflow на собственном сервере, вам не придется думать об администрировании Airflow.

Подобно прочим решениям о самостоятельной разработке или покупке продукта, упомянутым в этой книге, выбор между самостоятельным развертыванием Airflow или внешним управляемым размещением зависит от вашей конкретной ситуации:

- Есть ли у вас команда по системным операциям, которая поможет вам организовать самостоятельное обслуживание?
- Есть ли у вас средства на оплату управляемого размещения?
- Сколько DAG и задач содержат ваши конвейеры? Достаточно ли большой у вас масштаб, чтобы требовать более сложных исполнителей Airflow?
- Каковы ваши требования к безопасности и конфиденциальности? Допустимо ли позволять внешней службе подключаться к вашим внутренним данным и системам?

Другие фреймворки для оркестровки

Хотя настоящая глава посвящена Airflow, это отнюдь не единственный доступный вариант. Есть и другие отличные фреймворки для оркестровки, такие как Luigi и Dagster. Kubeflow Pipelines, ориентированный на оркестровку конвейеров машинного обучения, также хорошо поддерживается и популярен в сообществе машинного обучения.

Когда дело доходит до оркестровки шага преобразования для моделей данных, отличным выбором будет dbt от Fishtown Analytics. Как и Airflow, это продукт с открытым исходным кодом, созданный на Python, поэтому вы можете запускать его самостоятельно бесплатно или заплатить за управляемую облачную версию, которая называется dbt Cloud. Некоторые организации предпочитают применять Airflow или другой общий оркестратор для сбора данных и запуска таких вещей, как задания Spark, но затем используют dbt для преобразования своих моделей данных. В таком случае выполнение задания dbt запускается задачей в DAG Airflow, при этом dbt самостоятельно обрабатывает зависимости между моделями данных. Некоторые примеры использования dbt рассмотрены в *главе 9*.

ГЛАВА 8

Проверка данных в конвейерах

Даже в самом лучшем конвейере данных обязательно что-то пойдет не так. Многих проблем можно избежать или, по крайней мере, смягчить их, если правильно спроектировать процессы, оркестровку и инфраструктуру. Однако для обеспечения качества и достоверности самих данных вам необходимо озаботиться их проверкой. Лучше исходить из предположения, что непроверенные данные небезопасны для использования в аналитике. В этой главе обсуждаются принципы проверки данных на всех этапах конвейера ELT.

Проверяйте раньше, проверяйте чаще

Несмотря на благие намерения, некоторые команды специалистов по данным оставляют проверку на конец конвейера и впервые проверяют данные только во время преобразования или даже после завершения всех преобразований. Их подход основан на идее, что аналитики данных (которые обычно владеют логикой преобразования) лучше всего подходят для осмысления данных и определения наличия каких-либо проблем с качеством.

В такой схеме инженеры данных сосредоточены только на перемещении данных из одной системы в другую, организации конвейеров и обслуживании инфраструктуры данных. Хотя в этом и заключается роль инженера данных, в таком подходе кроется важный изъян: игнорируя содержимое данных, проходящих через каждый этап конвейера, они доверяют владельцам исходных систем, из которых они получают данные, а также своим собственным процессам сбора данных и аналитикам, преобразующим данные. Каким бы эффективным ни казалось такое разделение обязанностей, оно, скорее всего, приведет к низкому качеству

данных и неэффективному процессу отладки, когда будут обнаружены проблемы с качеством.

Обнаружение проблемы с качеством данных в конце конвейера и необходимость отследить ее до начала — это наихудший сценарий. Проверяя данные на каждом этапе конвейера, вы, скорее всего, своевременно обнаружите неполадку не возвращаясь к предыдущим этапам.

Хотя нельзя ожидать, что у инженеров данных будет достаточно контекста для проверки каждой выборки, они могут взять на себя инициативу, написав неконтекстные проверки, а также предоставив инфраструктуру и шаблоны, чтобы остальные члены команды, владельцы данных и прочие заинтересованные стороны могли принять участие в более специфических проверках на каждом этапе конвейера.

Качество данных исходной системы

Поскольку в типичной архитектуре существует большое количество исходных систем, поставляющих данные в хранилище, вполне вероятно, что в какой-то момент туда попадут недопустимые данные. Хотя может показаться, что владелец исходной системы должен был обнаружить некачественные данные до того, как они будут собраны, часто это не так по нескольким причинам:

- Неверные данные не всегда влияют на функционирование самой исходной системы. Логика исходного системного приложения может обходить такие проблемы, как повторяющиеся/неоднозначные записи в таблице, путем дедупликации на уровне приложения или заполнять нулевые значения даты значением по умолчанию в самом приложении.
- Исходная система может нормально функционировать, когда записи потеряны. Например, запись о клиенте может быть удалена, но записи о заказе, связанные с клиентом, могут остаться. Хотя приложение может просто игнорировать такие записи, эта ситуация, безусловно, повлияет на анализ данных.
- В исходной системе вполне может существовать ошибка, которая еще не найдена или не исправлена.

За свою карьеру я сталкивался с несколькими случаями, когда критическая проблема в исходной системе была выявлена именно командой обработки данных!

ПРИМЕЧАНИЕ

Независимо от причины, инженер данных никогда не должен предполагать, что данные, которые он собирает, не имеют проблем с качеством, даже если результирующие данные, загруженные в хранилище, идеально прошли через конвейер и полностью соответствуют их источнику.

Риски процесса сбора данных

Помимо проблем с качеством в исходной системе, существует вероятность того, что сам процесс сбора данных может привести к ухудшению качества данных. Вот несколько распространенных примеров:

- Сбой системы или тайм-аут на этапе извлечения или загрузки. Хотя иногда такая ситуация вызывает серьезную ошибку и останавливает конвейер, в других случаях "тихий" сбой приводит к частично извлеченному или загруженному набору данных.
- Логическая ошибка в инкрементном сборе данных. Вспомните, что было сказано в *главах 4* и *5* про шаблон для инкрементного извлечения. Сначала считывается временна́я метка самой последней записи из таблицы в хранилище данных, а затем извлекаются все записи с более поздней временно́й меткой в исходной системе, чтобы их можно было загрузить в хранилище. Даже такая простая логическая ошибка, как указание условия "больше или равно" вместо "больше" в операторе SQL, может привести к тому, что будут обработаны повторяющиеся записи. Существует множество других потенциальных источников ошибок, таких как несоответствие часовых поясов в разных системах.
- Проблемы парсинга файлов с данными. Как было сказано в *главах 4* и *5*, обычно данные извлекаются из исходной системы, сохраняются в плоском файле, таком как CSV, а затем загружаются из этого файла в хранилище данных. В данных, которые переводятся из исходной системы в неструктуриро-

ванный файл, иногда встречаются специальные символы или часть данных может быть представлена в неожиданной кодировке. В зависимости от того, как инженер данных и механизм загрузки хранилища данных обрабатывают такие случаи, записи могут быть удалены или данные, содержащиеся во вновь загруженных записях, так и останутся искаженными.

ПРИМЕЧАНИЕ

Оба предположения о том, что сбор данных "просто" извлекает и загружает данные, а также о том, что исходные системы будут предоставлять достоверные данные, неверны.

Проверка данных с участием аналитиков

Когда дело доходит до проверки данных, которые были загружены в хранилище, и данных, которые были преобразованы в модели, эту задачу лучше уступить аналитикам. Именно они понимают бизнес-контекст необработанных данных, а также каждой модели данных (см. *главу 6*). Инженеры данных должны предоставить аналитикам инструменты, необходимые им для определения и выполнения проверки данных по всему конвейеру. Конечно, проведение менее специфических проверок на ранних этапах конвейера, таких как соответствие числа строк и отсутствие повторяющихся записей, по-прежнему остается за инженерами данных.

В следующем разделе рассмотрен упрощенный фреймворк, который аналитики и инженеры данных могут использовать для реализации проверок достоверности данных в конвейере. В последнем разделе упоминаются несколько открытых и коммерческих фреймворков, которые подойдут для той же цели. Какой бы инструмент вы ни выбрали, важно предоставить инженерам и аналитикам надежный метод написания и выполнения проверочных тестов с минимальными затруднениями. Хотя на словах все члены команды обработки и анализа данных согласны с тем, что достоверные данные важны, если процедура проверки окажется сложной и трудоемкой, она быстро отойдет на второй план по сравнению с новыми разработками и другими приоритетами.

Простой фреймворк проверки данных

В этом разделе я опишу полнофункциональный фреймворк для проверки данных, написанный на Python и предназначенный для проверки данных с помощью SQL-запросов. Как и другие примеры в этой книге, он очень упрощен, и в нем отсутствуют многие функции, которые понадобятся в производственной среде. Другими словами, он не предназначен для удовлетворения всех ваших потребностей в проверке данных. Однако моя цель состоит в том, чтобы представить ключевые концепции такого фреймворка, а также поделиться чем-то, что можно расширить и улучшить в соответствии с вашей инфраструктурой.

Эта простая версия фреймворка имеет ограниченные возможности в отношении того, какие типы данных можно проверить и как выполняются пакетные тесты, но не более того. На случай, если вы захотите выбрать этот фреймворк в качестве отправной точки, позже я расскажу про некоторые возможные дополнения. Даже если вы решите взять готовый фреймворк, я считаю, что имеет смысл изучить ключевые идеи, связанные с этим очень упрощенным подходом.

Простой фреймворк проверки данных

Общая идея этой платформы для проверки данных (валидатора) — скрипт Python, который выполняет пару скриптов SQL и сравнивает их результаты с помощью оператора сравнения. Комбинация каждого скрипта и результата является *проверочным тестом*, и считается, что тест пройден или не пройден в зависимости от того, насколько результат выполненных скриптов соответствует ожидаемому результату. Например, один скрипт может подсчитывать число строк в таблице за определенный день, второй подсчитывает число строк за предыдущий день, а оператор сравнения >= проверяет, больше ли строк в текущем дне, чем в предыдущем. Если это так, тест пройден, если нет, тест провален.

Обратите внимание, что один из скриптов SQL также может возвращать статическое значение, например целое число. Как видно из кода в *разделе "Примеры проверок"* данной главы, этот подход применяется для проверки наличия повторяющихся строк в таб-

лице. Несмотря на свою простоту, этот фреймворк может обрабатывать широкий спектр логики проверки.

С помощью аргументов командной строки вы можете поручить валидатору выполнить определенную пару скриптов, а также задать оператор для сравнения результатов. Затем скрипт выполняется и возвращает результат прохождения теста. Имея возвращаемое значение, можно запустить различные действия в DAG Airflow, как показано далее в этом разделе, или реализовать любой другой процесс, запускаемый валидатором.

В листинге 8.1 показан код валидатора. Эта версия предназначена для выполнения тестов хранилища данных Amazon Redshift с использованием библиотеки Python psycopg2. Валидатор задействует тот же файл конфигурации pipe.conf из *глав 4* и *5* для доступа к учетным данным в хранилище. Вы можете легко изменить этот скрипт, чтобы получить доступ к хранилищу данных Snowflake согласно примерам из *главы 5* или к другому хранилищу данных по вашему выбору. Единственная разница будет заключаться в библиотеке, которая вам понадобится для подключения и выполнения запросов. Вам также необходимо убедиться, что ваша среда Python настроена правильно и виртуальная среда активирована перед запуском скрипта. Дополнительную информацию см. в *разделе "Настройка среды Python" главы 4*.

Листинг 8.1. Код упрощенного фреймворка в файле validator.py

```
import sys
import psycopg2
import configparser

def connect_to_warehouse():
    # Получение параметров подключения
    parser = configparser.ConfigParser()
    parser.read("pipeline.conf")
    dbname = parser.get("aws_creds", "database")
    user = parser.get("aws_creds", "username")
    password = parser.get("aws_creds", "password")
    host = parser.get("aws_creds", "host")
    port = parser.get("aws_creds", "port")
```

```
    rs_conn = psycopg2.connect(
        "dbname=" + dbname
        + " user=" + user
        + " password=" + password
        + " host=" + host
        + " port=" + port)

    return rs_conn

# Выполнение теста в виде двух скриптов
# и оператора сравнения
# Возвращает true/false, если тест пройден/провален
def execute_test(
        db_conn,
        script_1,
        script_2,
        comp_operator):

    # Выполнение скрипта 1 и сохранение результата
    cursor = db_conn.cursor()
    sql_file = open(script_1, 'r')
    cursor.execute(sql_file.read())

    record = cursor.fetchone()
    result_1 = record[0]
    db_conn.commit()
    cursor.close()

    # Выполнение скрипта 2 и сохранение результата
    cursor = db_conn.cursor()
    sql_file = open(script_2, 'r')
    cursor.execute(sql_file.read())

    record = cursor.fetchone()
    result_2 = record[0]
    db_conn.commit()
    cursor.close()

    print("result 1 = " + str(result_1))
    print("result 2 = " + str(result_2))
```

```
    # Сравнение результатов при помощи comp_operator
    if comp_operator == "equals":
        return result_1 == result_2
    elif comp_operator == "greater_equals":
        return result_1 >= result_2
    elif comp_operator == "greater":
        return result_1 > result_2
    elif comp_operator == "less_equals":
        return result_1 <= result_2
    elif comp_operator == "less":
        return result_1 < result_2
    elif comp_operator == "not_equal":
        return result_1 != result_2

    # Если мы здесь, что-то пошло не так
    return False

if __name__ == "__main__":

    if len(sys.argv) == 2 and sys.argv[1] == "-h":
        print("Использование: python validator.py"
          + "script1.sql script2.sql "
          + "comparison_operator")
        print("Допустимые значения comparison_operator:")
        print("equals")
        print("greater_equals")
        print("greater")
        print("less_equals")
        print("less")
        print("not_equal")
        exit(0)

    if len(sys.argv) != 4:
        print("Запуск: python validator.py"
          + "script1.sql script2.sql "
          + "comparison_operator")
        exit(-1)

    script_1 = sys.argv[1]
    script_2 = sys.argv[2]
    comp_operator = sys.argv[3]
```

```
# Подключение к хранилищу данных
db_conn = connect_to_warehouse()

# Выполнение проверочного теста
test_result = execute_test(
                db_conn,
                script_1,
                script_2,
                comp_operator)

print("Результат: " + str(test_result))

if test_result == True:
    exit(0)
else:
    exit(-1)
```

В следующих подразделах описывается структура проверочных тестов, для запуска которых предназначен этот фреймворк, и способы запуска теста из командной строки, а также из DAG Airflow. В следующем разделе я поделюсь некоторыми примерами проверочных тестов, основанных на распространенных типах тестирования.

Структура проверочного теста

Как кратко упоминалось в предыдущем подразделе, проверочный тест во фреймворке, приведенном в листинге 8.1, состоит из трех частей:

- файл SQL, который содержит первый скрипт, возвращающий первое числовое значение;
- файл SQL, который содержит второй скрипт, возвращающий первое числовое значение;
- "оператор сравнения", который предназначен для сравнения двух значений, возвращаемых скриптами SQL.

Рассмотрим простой пример теста, проверяющего, имеют ли две таблицы одинаковое число строк. В листинге 8.2 сценарий SQL подсчитывает число строк в таблице с именем `Orders`, а в листин-

ге 8.3 сценарий SQL получает то же число строк из другой таблицы с именем `Orders_Full`.

Листинг 8.2. Код в файле order_count.sql

```
SELECT COUNT(*)
FROM Orders;
```

Листинг 8.3. Код в файле order_full_count.sql

```
SELECT COUNT(*)
FROM Orders_Full;
```

Для создания и заполнения таблиц `Orders` и `Orders_Full` в примерах в этой главе вы можете использовать следующий SQL-запрос:

```
CREATE TABLE Orders (
  OrderId int,
  OrderStatus varchar(30),
  OrderDate timestamp,
  CustomerId int,
  OrderTotal numeric
);

INSERT INTO Orders VALUES(1,'Отгружен','2020-06-09',100,50.05);
INSERT INTO Orders VALUES(2,'Отгружен','2020-07-11',101,57.45);
INSERT INTO Orders VALUES(3,'Отгружен','2020-07-12',102,135.99);
INSERT INTO Orders VALUES(4,'Отгружен','2020-07-12',100,43.00);

CREATE TABLE Orders_Full (
  OrderId int,
  OrderStatus varchar(30),
  OrderDate timestamp,
  CustomerId int,
  OrderTotal numeric
);

INSERT INTO Orders_Full
  VALUES(1,'Отгружен ','2020-06-09',100,50.05);
INSERT INTO Orders_Full
  VALUES(2,'Отгружен ','2020-07-11',101,57.45);
```

```
INSERT INTO Orders_Full
  VALUES(3,'Отгружен ','2020-07-12',102,135.99);
INSERT INTO Orders_Full
  VALUES(4,'Отгружен ','2020-07-12',100,43.00);
```

Последней частью проверочного теста является оператор сравнения, который будет осуществлять сравнение двух значений. В образце кода из листинга 8.1 вы можете увидеть варианты выбора, доступные для операторов сравнения, но здесь они указаны с соответствующими логическими символами в круглых скобках для справки:

- `equals (==)`;
- `greater_equals (>=)`;
- `greater (>)`;
- `less_equals (<=)`;
- `less (<)`;
- `not_equal (!=)`.

Далее будет показано, как запустить тест и истолковать результат.

Запуск проверочного теста

Используя пример теста из предыдущего подраздела, проверку можно выполнить в командной строке следующим образом:

```
$ python validator.py order_count.sql
order_full_count.sql equals
```

Если число строк в таблицах `Orders` и `Orders_Full` совпадает, вывод будет выглядеть следующим образом:

```
result 1 = 15368
result 2 = 15368
Результат теста: True
```

Чего вы не видите в командной строке, так это *выходного кода состояния*, который в данном случае равен `0`, но будет равен `-1` в случае проваленного теста. Однако вы можете использовать это значение программно. В следующем разделе показано, как это сделать в DAG Airflow. Вы также можете организовать что-то

вроде отправки сообщения Slack или письма по электронной почте, когда тест не пройден. Я рассмотрю некоторые практические варианты позже в *разделе "Дополнения к фреймворку"* данной главы.

Использование фреймворка в DAG Airflow

Как было сказано в *главе 7*, задача Airflow может выполнять скрипт Python с помощью команды `BashOperator`. Возьмем DAG `elt_pipeline_sample` из листинга 7.2. После того как таблица `Orders` будет получена (после задач извлечения и загрузки), я добавлю еще одну задачу для запуска примера проверочного теста, которым я только что с вами поделился, чтобы сравнить число строк таблицы `Orders` с некоторой вымышленной таблицей с именем `Orders_Full`. Ради этого примера предположим, что по какой-то причине мы хотим убедиться, что число строк в `Orders` совпадает с `Orders_Full`, и если это не так, то отметить сбой выполнения задачи и остановить дальнейшее выполнение нижестоящих задач в DAG.

Сначала добавьте следующую задачу в определение DAG в файле elt_pipeline_sample.py:

```
check_order_rowcount_task = BashOperator(
    task_id='check_order_rowcount',
    bash_command='set -e; python validator.py' +
    'order_count.sql order_full_count.sql equals',
    dag=dag,
)
```

Затем переопределите порядок зависимостей DAG в том же файле, указав приведенный далее код. Он устроен так, что после `load_orders_task` запустится задача проверки, а затем будет запущена задача `revenue_model_task`, как только проверка будет завершена (и пройдена), а также успешно завершится задача `load_customers_task`:

```
extract_orders_task >> load_orders_task
extract_customers_task >> load_customers_task
load_orders_task >> check_order_rowcount_task
check_order_rowcount_task >> revenue_model_task
load_customers_task >> revenue_model_task
```

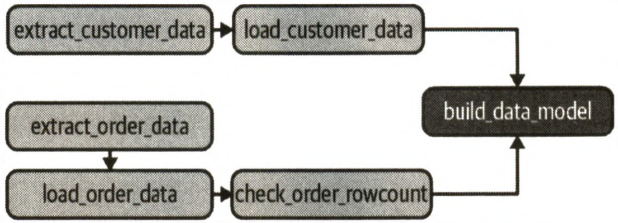

Рис. 8.1. Графовое представление примера ELT DAG с добавленным тестом

На рис. 8.1 показано обновленное графовое представление DAG.

Когда выполняется `check_order_rowcount_task`, то в соответствии с определением задачи запускается следующая команда Bash:

```
set -e; python validator.py order_count.sql
order_full_count.sql equals
```

Здесь нетрудно узнать запуск валидатора с помощью аргументов командной строки, показанный ранее в этом разделе. Но здесь появилась опция `set -e;` перед остальной частью команды. Она говорит Bash остановить выполнение скрипта при ошибке, которая определяется ненулевым кодом состояния выхода. Как вы помните, если проверочный тест не пройден, он возвращает состояние выхода `-1`. Если это произойдет, задача Airflow завершится ошибкой, и никакие последующие задачи не будут выполняться (в данном случае — `revenue_model_task`).

Не всегда нужно останавливать дальнейшее выполнение DAG, если проверочные тесты не пройдены. В таком случае вам следует убрать часть `set -e` набора команд Bash из запуска задачи Airflow или изменить валидатор для раздельной обработки предупреждений и серьезных ошибок. Далее я расскажу, когда это следует делать, а когда лучше просто отправить какое-то уведомление.

Когда нужно остановить конвейер, а когда предупредить и продолжить

Бывают случаи, когда необходимо остановить конвейер, если тесты проверки данных провалены. В текущем примере, если число записей в таблице `Orders` неправильное, возможно, из-за обновления модели данных в последней задаче, бизнес-пользо-

ватели увидят неверные сведения о продажах. Если этого важно избежать, то правильным решением будет остановить DAG, чтобы решить проблему. Когда это будет сделано, в модели данных все еще будут содержаться данные предыдущего успешного запуска DAG. В большинстве случаев устаревшие данные лучше неверных!

Однако бывают и другие случаи, когда провал проверочного теста менее критичен и более информативен. Например, возможно, количество заказов в таблице увеличилось на 3% по сравнению с предыдущим запуском день назад, в то время как средний дневной прирост за предыдущие 30 дней составил 1%. Вы можете перехватить такое увеличение с помощью базового статистического теста, как я покажу в следующем разделе. Нужно ли останавливать выполнение в такой ситуации? Ответ заключается в том, что это зависит от ваших обстоятельств и склонности к риску, но вы можете запустить несколько разных тестов, чтобы получить окончательный ответ.

Например, если вы также запустили тест на наличие повторяющихся строк в таблице Orders и он прошел успешно, то вы знаете, что проблема не в каком-то дублировании. Возможно, у компании просто был невероятно удачный день продаж из-за рекламной акции. Вы также можете настроить свой тест, чтобы учесть сезонность. Возможно, сейчас сезон отпусков, а вчера была "черная пятница". Вместо того, чтобы сравнивать рост числа записей с предыдущими 30 днями, вы должны были сравнить его с тем же периодом прошлого года (с учетом коэффициента роста или без него).

В целом решение об ошибке и остановке конвейера или отправке оповещения в канал Slack должно основываться на контексте бизнеса и сценарии использования данных. А еще это говорит о том, что не только инженеры данных, но и аналитики должны иметь возможность добавлять проверочные тесты в конвейер. Хотя инженер данных может проверить несоответствие числа строк, он не обязательно будет знать все нюансы бизнес-контекста, чтобы догадаться учесть фактор сезонности в росте количества строк таблицы `Orders`.

Что если вам нужно просто выдать предупреждение, а не останавливать конвейер? Для этого потребуется внести несколько изменений либо в DAG из предыдущего примера, либо в сам

фреймворк. В Airflow есть несколько вариантов обработки ошибок, о которых вы можете узнать в официальной документации Airflow. В следующем разделе, посвященном некоторым дополнениям к фреймворку валидации, я предлагаю несколько способов обработки менее критических сбоев в самом фреймворке. Любой вариант будет правильным; вам решать, где размещать логику проверки.

Дополнения к фреймворку

Как я уже отмечал ранее в этой главе, в коде фреймворка проверки данных из листинга 8.1 отсутствуют многие функции, которые вам наверняка понадобятся при развертывании в рабочей среде. Если вы решите выбрать в качестве отправной точки именно этот фреймворк, а не приложение с открытым исходным кодом или коммерческий продукт, то у вас есть возможность добавить в него несколько расширений.

Общепринятая функция средств проверки — отправка уведомления на канал Slack или по электронной почте в случае провала теста. Я приведу пример того, как это сделать для канала Slack, но в Интернете есть множество примеров отправки электронной почты и уведомлений в другие службы обмена сообщениями из скрипта Python.

Во-первых, вам нужно создать *веб-перехватчик входящих сообщений* (incoming webhook) для канала Slack, в который вы хотите отправлять сообщения. Веб-перехватчик — это уникальный для канала URL-адрес, на который вы можете публиковать данные, чтобы они отображались в виде сообщения на этом канале. Обратитесь к руководству в документации Slack, чтобы узнать, как его создать.

После создания веб-перехватчика добавьте в файл validator.py функцию, показанную в листинге 8.4. Вы можете передать ей информацию о проверочном тесте. Информация, отправленная на веб-перехватчик, затем публикуется в канале Slack.

Листинг 8.4. Функция для отправки сообщений в канал Slack

```
# test_result должен иметь значение True/False
def send_slack_notification(
  webhook_url,
```

```
  script_1,
  script_2,
  comp_operator,
  test_result):
    try:
        if test_result == True:
            message = ("Проверка пройдена!: "
            + script_1 + " / "
            + script_2 + " / "
            + comp_operator)
        else:
            message = ("Проверка ПРОВАЛЕНА!: "
            + script_1 + " / "
            + script_2 + " / "
            + comp_operator)

        slack_data = {'text': message}
        response = requests.post(webhook_url,
            data=json.dumps(slack_data),
            headers={
                'Content-Type': 'application/json'
            })

        if response.status_code != 200:
            print(response)
            return False
    except Exception as e:
        print("Ошибка отправки сообщения slack")
        print(str(e))
        return False
```

Теперь все, что вам нужно сделать, — это вызвать функцию прямо перед выходом из validation.py. В листинге 8.5 показаны последние строки обновленного скрипта.

Листинг 8.5. Отправка сообщения Slack, когда тест провален

```
if test_result == True:
        exit(0)
    else:
        send_slack_notification(
          webhook_url,
```

```
            script_1,
            script_2,
            comp_operator,
            test_result)
        exit(-1)
```

Конечно, можно добавить форматирование сообщений Slack, которые отправляет функция, но пока этого достаточно для выполнения основной работы. Обратите внимание, что я добавил параметр `test_result` в функцию `send_slack_notification`. Он настроен для обработки уведомлений как о пройденных тестах, так и о проваленных. При желании вы можете отправлять два типа сообщений, хотя я в своем примере так не делаю.

Как было сказано в предыдущем подразделе, иногда достаточно сообщения Slack, и провал теста не должен приводить к остановке конвейера. Для обработки такого случая вы можете использовать конфигурацию DAG или усовершенствовать фреймворк, добавив еще один параметр командной строки для определения серьезности проблемы.

В листинге 8.6 показан обновленный блок `__main__` скрипта validator.py с обработкой уровня серьезности. Когда скрипт выполняется с уровнем серьезности `halt` (останов), в случае провала теста выдается код выхода `-1`. Если задан уровень серьезности `warn` (предупреждение), то при неудачном завершении теста код выхода будет равен `0`, как и в случае успешного прохождения теста. В обоих случаях провал теста приводит к отправке сообщения Slack в нужный канал.

Листинг 8.6. Добавление обработки нескольких уровней серьезности сбоя

```
if __name__ == "__main__":

    if len(sys.argv) == 2 and sys.argv[1] == "-h":
        print("Использование: python validator.py"
            + "script1.sql script2.sql "
            + "comparison_operator")
        print("Допустимые значения comparison_operator:")
        print("equals")
```

```
        print("greater_equals")
        print("greater")
        print("less_equals")
        print("less")
        print("not_equal")

        exit(0)

    if len(sys.argv) != 5:
        print("Использование: python validator.py"
            + "script1.sql script2.sql "
            + "comparison_operator")
        exit(-1)

    script_1 = sys.argv[1]
    script_2 = sys.argv[2]
    comp_operator = sys.argv[3]
    sev_level = sys.argv[4]

    # Подключение к хранилищу данных
    db_conn = connect_to_warehouse()

    # Выполнение теста
    test_result = execute_test(
                  db_conn,
                  script_1,
                  script_2,
                  comp_operator)

    print("Результат теста: " + str(test_result))

    if test_result == True:
        exit(0)
    else:
        send_slack_notification(
          webhook_url,
          script_1,
          script_2,
          comp_operator,
          test_result)
        if sev_level == "halt":
            exit(-1)
```

```
    else:
        exit(0)
```

Есть бесчисленное множество способов дополнить фреймворк, два из которых приведены далее. Я уверен, вы придумаете и другие!

- Обработка исключений через приложение. Хотя я не привожу код для экономии места в этой книге, перехват и обработка исключений для таких событий, как недопустимые аргументы командной строки и ошибки SQL в тестовых сценариях, являются обязательными в производственной среде.
- Возможность запуска нескольких тестов одним выполнением validator.py. Подумайте о том, чтобы сохранить свои тесты в файле конфигурации и сгруппировать их по таблице с помощью DAG или другим способом, который соответствует вашему шаблону разработки. Затем вы можете выполнить все тесты, соответствующие определенной точке конвейера, задав одну команду, а не запуская каждый тест в отдельности.

Примеры проверок

В предыдущем разделе был показан простой фреймворк проверки и концепция его работы. Напомню, что проверочный тест состоит из следующих компонентов:

- файл SQL, который запускает скрипт, возвращающий одно числовое значение;
- второй файл SQL, который запускает скрипт, возвращающий одно числовое значение;
- оператор сравнения, который предназначен для сравнения двух значений, возвращаемых скриптами SQL.

Далее я предполагаю, что вы добавили доработки из листингов 8.4–8.6 в код validator.py, приведенный в листинге 8.1. Теперь вы можете запустить тест из командной строки следующим образом:

```
python validator.py order_count.sql
order_full_count.sql equals warn
```

Уровень серьезности в примерах

Обратите внимание, что я использую значение `warn` для последнего параметра командной строки (`severity_level`), как показано выше. Я буду делать это во всех примерах данного раздела, но вы также можете задать значение `halt`, если хотите. Обратитесь к листингу 8.6 для получения дополнительной информации.

В этом разделе я продемонстрирую несколько тестов, которые пригодятся для проверки данных в конвейере. Это далеко не все тесты, которые вам придется выполнять в будущем, но они охватывают некоторые общие моменты, важные для успешного начала работы, и вдохновят вас на более широкий спектр проверок. Каждый подраздел включает исходный код двух файлов SQL, составляющих тест, а также команды командной строки и аргументы для выполнения тестов.

Дубликаты записей после сбора данных

Проверка дубликатов записей — это простой и распространенный тест. Единственное, нужно заранее подумать, что будет служить дубликатом в таблице, которую вы проверяете. Это будет только значение идентификатора? Идентификатор, а также второй столбец? В следующем примере (листинг 8.7) я проверю, чтобы в таблице `Orders` не было двух записей с одинаковым `OrderId`. Чтобы проверить дубликаты на основе дополнительных столбцов, вы можете просто добавить эти столбцы в `SELECT` и `GROUP BY` в первом запросе.

Обратите внимание, что второй запрос возвращает статическое значение `0` (листинг 8.8). Дело в том, что я не допускаю наличия дубликатов, поэтому хочу сравнить число дубликатов с нулем. Если так и есть, тест пройден.

Листинг 8.7. Код скрипта из файла order_dup.sql

```
WITH order_dups AS
(
  SELECT OrderId, Count(*)
  FROM Orders
  GROUP BY OrderId
  HAVING COUNT(*) > 1
)
```

```
SELECT COUNT(*)
FROM order_dups;
```

Листинг 8.8. Код скрипта из файла order_dup_zero.sql

```
SELECT 0;
```

Для запуска теста выполните следующую команду:

```
python validator.py order_dup.sql
order_dup_zero.sql equals warn
```

Неожиданное изменение числа строк после сбора данных

Если вы уверены, что число записей, полученных при сборе данных, должно быть относительно постоянным, то можете использовать статистическую проверку, чтобы увидеть, загружено ли при последнем добавлении больше или меньше записей, чем предполагает история предыдущих загрузок.

В этом примере я предполагаю, что данные загружаются ежедневно, и буду проверять, находится ли число записей в таблице `Orders`, загруженной последними (вчера), в удобном для меня диапазоне. Вы можете сделать то же самое для часового, недельного или любого другого интервала, если он не меняется.

Я воспользуюсь расчетом стандартного отклонения и посмотрю, находится ли вчерашнее число строк в пределах 90%-ного доверительного уровня на основе всей истории таблицы `Orders`. Другими словами, находится ли фактическое значение (число строк) в пределах 90% доверительного интервала в любом направлении (может отличаться до 5% в любом направлении) от ожидаемого, основанного на истории?

Насколько далеко нужно оглядываться назад

Вы можете оглянуться на меньший период времени, чем вся история таблицы, например на год или два. Это решение должно быть основано на истории данных. Были ли систематические изменения в какой-то момент времени? Верна ли история более чем годичной давности? Это решение зависит от вас.

В статистике это считается *проверкой с двусторонним критерием*, потому что мы рассматриваем обе стороны кривой нормального распределения. Вы можете использовать калькулятор *z*-показателя, чтобы определить, какой показатель выбрать для двустороннего теста с доверительным интервалом 90% в случае *z*-показателя, равного 1,645. Другими словами, мы ищем разницу, выходящую за установленный порог в любом направлении.

Я буду применять в тесте этот *z*-показатель, чтобы увидеть, проходит ли тест число записей о заказах со вчерашнего дня. В проверочном тесте я верну абсолютное значение *z*-оценки для вчерашнего числа строк, а затем сравню ее с *z*-оценкой 1,645 во втором SQL-скрипте.

Поскольку для работы этого теста требуется большое количество образцов данных в таблицах `Orders`, я привожу две версии первого SQL-скрипта для проверки. Первая (листинг 8.9) — это "настоящий" код, используемый для просмотра таблицы `Orders`, получения числа строк по дням и последующего вычисления *z*-показателя за предыдущий день.

Возможно, вам захочется взять готовые образцы данных для экспериментов с таким тестом. На этот случай я предоставляю альтернативную версию для заполнения таблицы с именем `orders_by_day`, а затем выполняю последний раздел листинга 8.9, чтобы вычислить *z*-значение для последнего дня выборки (2020-10-05). В листинге 8.11 приведена альтернативная версия.

Листинг 8.9. Код скрипта из файла order_yesterday_zscore.sql

```
WITH orders_by_day AS (
  SELECT
    CAST(OrderDate AS DATE) AS order_date,
    COUNT(*) AS order_count
  FROM Orders
  GROUP BY CAST(OrderDate AS DATE)
),
order_count_zscore AS (
  SELECT
  order_date,
  order_count,
```

```
  (order_count - avg(order_count) over ())
   / (stddev(order_count) over ()) as z_score
  FROM orders_by_day
)
SELECT ABS(z_score) AS twosided_score
FROM order_count_zscore
WHERE
  order_date =
  CAST(current_timestamp AS DATE)
  - interval '1 day';
```

Пример из листинга 8.10 просто возвращает значение для проверки.

Листинг 8.10. Код скрипта из файла zscore_90_twoside.sql

```
SELECT 1.645;
```

Для запуска теста используйте следующую команду:

```
python validator.py order_yesterday_zscore.sql
zscore_90_twosided.sql greater_equals warn
```

> **ПРИМЕЧАНИЕ**
>
> Если таблица `Orders` содержит большой объем данных, имеет смысл создать набор данных `orders_by_day` как таблицу в задаче преобразования (точно так же, как примеры модели данных в *главе 6*), а не как CTE в сценарии проверки. Поскольку количество заказов по дням не должно меняться для прошедших дат, вы можете создать добавочную модель данных и добавлять строки для каждого последующего дня по мере поступления новых данных в таблицу `Orders`.

Вот альтернативная версия с жестко заданной датой для проверки вместе с примерами данных, необходимых для ее запуска. В этой версии вы можете настроить значения `order_count` и запустить тест, чтобы получить разные *z*-показатели в нужном диапазоне и за его пределами:

```
CREATE TABLE orders_by_day
(
  order_date date,
  order_count int
);
```

```
INSERT INTO orders_by_day VALUES ('2020-09-24', 11);
INSERT INTO orders_by_day VALUES ('2020-09-25', 9);
INSERT INTO orders_by_day VALUES ('2020-09-26', 14);
INSERT INTO orders_by_day VALUES ('2020-09-27', 21);
INSERT INTO orders_by_day VALUES ('2020-09-28', 15);
INSERT INTO orders_by_day VALUES ('2020-09-29', 9);
INSERT INTO orders_by_day VALUES ('2020-09-30', 20);
INSERT INTO orders_by_day VALUES ('2020-10-01', 18);
INSERT INTO orders_by_day VALUES ('2020-10-02', 14);
INSERT INTO orders_by_day VALUES ('2020-10-03', 26);
INSERT INTO orders_by_day VALUES ('2020-10-04', 11);
```

Листинг 8.11. Код скрипта из файла order_sample_zscore.sql

```
WITH order_count_zscore AS (
  SELECT
    order_date,
    order_count,
    (order_count - avg(order_count) over ())
     / (stddev(order_count) over ()) as z_score
  FROM orders_by_day
)
SELECT ABS(z_score) AS twosided_score
FROM order_count_zscore
WHERE
  order_date =
    CAST('2020-10-05' AS DATE)
    - interval '1 day';
```

Чтобы запустить тест, воспользуйтесь следующей командой:

```
python validator.py order_sample_zscore.sql
zscore_90_twosided.sql greater_equals warn
```

Колебания значения показателя

Как отмечалось ранее в этой главе, проверка данных на каждом этапе конвейера имеет решающее значение. В предыдущих двух примерах проверялась достоверность данных после получения. В следующем примере выполняется проверка после моделирования данных на этапе преобразования конвейера.

В примерах моделирования данных из *главы 6* несколько исходных таблиц объединены вместе и реализована логика, определяющая, как агрегировать значения. На этом этапе многие вещи могут пойти не так, включая неверную логику объединения, которая приводит к дублированию или удалению строк. Даже если исходные данные прошли проверку ранее в конвейере, всегда рекомендуется запускать проверку моделей данных, созданных в конце конвейера.

Есть три теста, которые вы можете выполнить на этом этапе:

- проверка того, находится ли определенный показатель между нижней и верхней границами;
- проверка увеличения (или уменьшения) числа строк в модели данных;
- проверка на наличие неожиданных колебаний значения конкретного показателя.

К настоящему моменту вы, вероятно, уже хорошо представляете, как реализовать такие тесты, но я приведу последний пример проверки колебаний значения показателя. Логика скрипта почти идентична логике из предыдущего раздела, где я рассказал, как использовать двусторонний тест для проверки изменения числа строк в заданной исходной таблице. Однако на этот раз вместо проверки значения числа строк я смотрю, не выходит ли общий доход от заказов, размещенных в данный день, за исторические нормы.

Аналогично примеру с поиском изменений числа строк в предыдущем разделе я привожу как "реальный" пример того, как это сделать с необработанными данными (листинг 8.12), так и пример с выборочными агрегированными данными (листинг 8.14). Чтобы выполнить пример из листинга 8.12, вам понадобится довольно много данных в таблице `Orders`. Этот код целесообразно применять в настоящей реализации, а с примером, приведенным в листинге 8.14, легче экспериментировать ради обучения.

Листинг 8.12. Код скрипта из файла revenue_yesterday_zscore.sql

```
WITH revenue_by_day AS (
  SELECT
    CAST(OrderDate AS DATE) AS order_date,
    SUM(ordertotal) AS total_revenue
```

```
  FROM Orders
  GROUP BY CAST(OrderDate AS DATE)
),
daily_revenue_zscore AS (
  SELECT
    order_date,
    total_revenue,
    (total_revenue - avg(total_revenue) over ())
     / (stddev(total_revenue) over ()) as z_score
  FROM revenue_by_day
)
SELECT ABS(z_score) AS twosided_score
FROM daily_revenue_zscore
WHERE
  order_date =
    CAST(current_timestamp AS DATE)
    - interval '1 day';
```

Пример 8.13 просто возвращает значение для проверки.

Листинг 8.13. Код скрипта из файла zscore_90_twoside.sql

```
SELECT 1.645;
```

Для запуска теста используйте следующую команду:

```
python validator.py revenue_yesterday_zscore.sql
zscore_90_twosided.sql greater_equals warn
```

Вот выборка данных для листинга 8.14, который, как отмечалось ранее, является упрощенной версией листинга 8.12, но для ваших собственных экспериментов:

```
CREATE TABLE revenue_by_day
(
  order_date date,
  total_revenue numeric
);

INSERT INTO revenue_by_day VALUES ('2020-09-24', 203.3);
INSERT INTO revenue_by_day VALUES ('2020-09-25', 190.99);
INSERT INTO revenue_by_day VALUES ('2020-09-26', 156.32);
```

```
INSERT INTO revenue_by_day VALUES ('2020-09-27', 210.0);
INSERT INTO revenue_by_day VALUES ('2020-09-28', 151.3);
INSERT INTO revenue_by_day VALUES ('2020-09-29', 568.0);
INSERT INTO revenue_by_day VALUES ('2020-09-30', 211.69);
INSERT INTO revenue_by_day VALUES ('2020-10-01', 98.99);
INSERT INTO revenue_by_day VALUES ('2020-10-02', 145.0);
INSERT INTO revenue_by_day VALUES ('2020-10-03', 159.3);
INSERT INTO revenue_by_day VALUES ('2020-10-04', 110.23);
```

Листинг 8.14. Код скрипта из файла revenue_sample_zscore.sql

```
WITH daily_revenue_zscore AS (
  SELECT
    order_date,
    total_revenue,
    (total_revenue - avg(total_revenue) over ())
     / (stddev(total_revenue) over ()) as z_score
  FROM revenue_by_day
)
SELECT ABS(z_score) AS twosided_score
FROM daily_revenue_zscore
WHERE
  order_date =
    CAST('2020-10-05' AS DATE)
    - interval '1 day';
```

Для запуска теста используйте следующую команду:

```
python validator.py revenue_sample_zscore.sql
zscore_90_twosided.sql greater_equals warn
```

Конечно, вам необходимо будет скорректировать этот тест в соответствии с вашими бизнес-процессами.

Не слишком ли хлопотно смотреть на выручку от заказов по дням? Возможно, объем ваших заказов достаточно мал, и целесообразнее проверять недельные или месячные сводные данные. Если это так, вы можете изменить листинг 8.12, чтобы агрегировать показатель по неделям или месяцам, а не по дням. В листинге 8.15 показана ежемесячная версия той же проверки. Она сравнивает показатель минувшего месяца с 11 предшествующими месяцами.

Обратите внимание, что в этом примере проверяется общий доход за прошедший месяц с текущей даты. Это тип проверки, которую вы запускаете, когда "закрываете" месяц, что обычно происходит в первый день следующего месяца. Например, это проверка, которую вы можете запустить 1 октября, чтобы убедиться, что доход за сентябрь находится в ожидаемом диапазоне на основе исторических данных.

Листинг 8.15. Код скрипта из файла revenue_lastmonth_zscore.sql

```
WITH revenue_by_day AS (
  SELECT
    date_part('month', order_date) AS order_month,
    SUM(ordertotal) AS total_revenue
  FROM Orders
  WHERE
    order_date > date_trunc('month',
                            current_timestamp - interval '12 months')
    AND
    order_date < date_trunc('month', current_timestamp)
  GROUP BY date_part('month', order_date)
),
daily_revenue_zscore AS (
  SELECT
    order_month,
    total_revenue,
    (total_revenue - avg(total_revenue) over ())
     / (stddev(total_revenue) over ()) as z_score
  FROM revenue_by_day
)
SELECT ABS(z_score) AS twosided_score
FROM daily_revenue_zscore
WHERE order_month =
date_part('month',date_trunc('month',
                            current_timestamp - interval '1 months'));
```

Существуют и другие варианты подобной проверки. Какой уровень детализации дат выбрать, какие периоды дат сравнить, и даже значение *z*-оценки — это параметры, которые вам нужно будет проанализировать и настроить на основе ваших собственных данных.

Проверка показателя требует знания контекста

Написание тестов для значений показателей в модели данных может быть довольно сложной задачей, и ее лучше оставить аналитику данных, хорошо знающему бизнес-контекст. Умение одновременно учитывать рост бизнеса, влияние дня недели, сезонности и т. п. само по себе является навыком и различается для каждого бизнеса и варианта применения. Тем не менее примеры в этом разделе должны дать вам представление о том, с чего начать.

Коммерческие и открытые фреймворки проверки данных

В этой главе я привел примеры применения проверочного фреймворка на основе Python. Как отмечалось ранее, несмотря на простоту, его можно легко расширить, чтобы он стал полнофункциональным, готовым к работе приложением для всех видов потребностей в проверке данных.

Тем не менее, как и в случае со сбором и моделированием данных и инструментами оркестровки данных, когда дело доходит до инструмента проверки, необходимо принять решение о разработке собственного или покупке готового продукта. Предшествующие решения о разработке или покупке часто влияют на то, какой инструмент группа работы с данными решает применить для проверки данных в разных точках конвейера.

Например, некоторые инструменты сбора данных включают функции проверки изменений числа строк, неожиданных значений в столбцах и т. д. Некоторые фреймворки преобразования данных, такие как dbt, включают проверку данных и функциональное тестирование. Если вы уже используете такие инструменты, проверьте, какие варианты проверок доступны.

Наконец, существуют фреймворки с открытым исходным кодом для проверки данных. Количество таких фреймворков огромно, и я предлагаю выбрать тот, который подходит для вашей экосистемы. Например, если вы строите конвейер машинного обучения и используете TensorFlow, то можете выбрать инструмент TensorFlow Data Validation (**https://www.tensorflow.org/tfx/data_validation/get_started**). Для более универсальной проверки подойдет Yahoo Validator с открытым исходным кодом.

Передовые методы обслуживания конвейеров

Предыдущие главы были посвящены построению конвейеров данных. В этой главе обсуждается, как поддерживать работающие конвейеры по мере того, как они становятся все более сложными, и как справляться с неизбежными изменениями в системах, на которые опираются ваши конвейеры.

Как реагировать на изменения в исходных системах

Одна из наиболее распространенных проблем обслуживания для инженеров данных связана с тем фактом, что системы, из которых они получают данные, не статичны. Разработчики постоянно вносят изменения в свое программное обеспечение, добавляя функции, выполняя рефакторинг кодовой базы или исправляя ошибки. Когда эта их деятельность приводит к изменению схемы или значения собираемых данных, возникает риск ошибки или остановки конвейера.

Как я уже говорил, реалии современной инфраструктуры данных таковы, что информация поступает из самых разных источников. В результате сложно найти универсальное решение для обработки изменений схемы и бизнес-логики в исходных системах. Тем не менее есть несколько проверенных методов, на которые я рекомендую обратить внимание.

Добавление абстракции

Если есть возможность, то между исходной системой и процессом сбора данных нужно ввести уровень абстракции. Также важно, чтобы владелец исходной системы либо поддерживал метод абстракции, либо знал о нем.

Например, вместо получения данных непосредственно из БД Postgres попробуйте договориться с владельцем базы данных о создании REST API, который извлекает данные из БД и может быть запрошен извне для запуска извлечения данных. Даже если API представляет собой просто сквозной канал, тот факт, что он существует в кодовой базе, поддерживаемой владельцем исходной системы, означает, что владелец системы знает, какие данные извлекаются, и ему не нужно беспокоиться об изменениях во внутренней структуре БД приложения Postgres. Если он решит изменить структуру таблицы БД, ему достаточно будет внести соответствующие изменения в API со своей стороны, и не придется решать проблему совместимости изменений с внешним кодом сбора данных.

Если изменение в исходной системе приводит к удалению поля, которое использует поддерживаемая конечная точка API, то может быть принято взвешенное решение относительно того, что делать. Возможно, это поле со временем прекратит свое существование или какое-то время будет поддерживаться историческими данными, а потом начнет возвращать значение `NULL`. В любом случае, когда существует явный уровень абстракции, существует и осознание необходимости обработки изменения.

REST API — не единственный вариант абстракции, а иногда и не лучший вариант. Публикация данных через топики Kafka — отличный способ поддерживать согласованную схему, при этом полностью отделяя друг от друга исходную систему, которая публикует событие, и систему, которая подписывается на него (для сбора данных).

Поддержка контрактов данных

Если вам необходимо получать данные непосредственно из БД исходной системы или применять какой-либо метод, в явном виде не предназначенный для сбора данных, то относительно несложным техническим решением будет создание и поддержка контракта данных.

Контракты данных

Контракт данных (data contract) — это письменное соглашение между владельцем исходной системы и командой специалистов, получающей

данные из этой системы для использования в конвейере данных. В контракте должно быть указано, какие данные извлекаются, каким методом (полным, инкрементным), как часто, а также кто (человек, команда) является контактным лицом как для исходной системы, так и для потребителя данных. Контракты данных должны храниться в известном и легкодоступном месте, например в репозитории GitHub или на сайте внутренней документации. Если возможно, отформатируйте контракты данных в стандартизированное представление, чтобы их можно было интегрировать в процесс разработки или запрашивать программно.

Контракт данных может быть записан в виде текстового документа, но лучше в виде стандартизированного файла конфигурации, такого как в листинге 9.1. В этом примере контракт данных для извлечения из таблицы в БД Postgres хранится в формате JSON.

Листинг 9.1. Контракт данных orders_contract.json

```
{
  ingestion_jobid: "orders_postgres",
  source_host: "my_host.com",
  source_db: "ecommerce",
  source_table: "orders",
  ingestion_type: "full",
  ingestion_frequency_minutes: "60",
  source_owner: "dev-team@mycompany.com",
  ingestion_owner: "data-eng@mycompany.com"
};
```

После того как контракты данных будут созданы, вы можете использовать их, чтобы опережать любые изменения исходной системы, которые могут поставить под угрозу целостность ваших конвейеров:

- создайте хук (ловушку) Git, которая ищет любые изменения схемы или логики в таблице, указанной как `source_table` в контракте данных, когда отправляется запрос на добавление (push request) или код прописывается в ветке. Должно срабатывать автоматическое уведомление инициатора изменения о том, что таблица используется при приеме данных, с указанием контактных данных (`ingestion_owner`) для координации изменений;

- если сам контракт данных находится в репозитории Git (а так и должно быть!), добавьте хук Git для проверки изменений в контракте. Например, если частота, с которой выполняется сбор данных, увеличивается, следует не только обновить контракт данных, но и проконсультироваться с владельцем исходной системы, чтобы убедиться, что это не оказывает негативного влияния на производственную систему;
- опубликуйте в удобочитаемой форме все контакты данных на централизованном сайте документации компании и сделайте их доступными для поиска;
- напишите специальный скрипт и запланируйте его выполнение таким образом, чтобы уведомлять владельцев исходной системы и потребителей данных о любых контрактах данных, которые не обновлялись в течение последних шести месяцев (или с другой периодичностью), и просить их проверить контракты и обновить их при необходимости.

Независимо от уровня автоматизации, цель состоит в том, чтобы изменения в принимаемых данных или в методе сбора данных (скажем, при переходе от инкрементной к полной загрузке) помечались и доводились до сведения участников процесса прежде, чем возникнут проблемы в конвейере или исходной системе.

Ограничения схемы при чтении

Один из подходов к обработке изменений схемы исходных данных состоит в том, чтобы перейти от *структурирования при записи* (schema-on-write) к *структурированию при чтении* (schema-on-read).

Структурирование при записи — это подход, используемый в данной книге, в частности в *главах 4* и *5*. Когда данные извлекаются из источника, определяется структура (схема) и данные записываются в озеро данных или корзину S3. К моменту, когда выполняется шаг загрузки, данные находятся в предсказуемой форме и могут быть загружены в таблицу с определенной структурой.

Структурирование при чтении — это подход, при котором данные записываются в озеро данных, корзину S3 или другую сис-

тему хранения без строгой схемы. Например, событие, определяющее размещенный в системе заказ, может быть определено как объект JSON, но структура этого объекта может меняться со временем по мере добавления новых или удаления существующих свойств. В этом случае структура данных остается неизвестной до тех пор, пока они не будут прочитаны, поэтому она называется структурой при чтении.

Хотя описанный подход очень эффективен для записи данных в хранилище, он усложняет этап загрузки и имеет некоторые важные последствия для конвейера. С технической точки зрения чтение данных, хранящихся таким образом, из корзины S3 не вызывает особых затруднений. Amazon Athena и другие продукты делают запросы к необработанным данным такими же простыми, как написание SQL-запроса. Однако обслуживание определения данных — непростая задача.

Во-первых, вам потребуется *каталог данных*, который интегрируется с любым инструментом, применяемым для чтения данных с гибкой схемой на этапе загрузки. В каталоге данных хранятся метаданные для данных в вашем озере и хранилище. Он может хранить как структуру, так и определение наборов данных. В случае структурирования при чтении очень важно определить и сохранить структуру данных в каталоге как для прикладного использования, так и для ознакомления людьми. Наиболее широко распространены каталоги данных AWS Glue Data Catalog и Apache Atlas, но выбор намного шире.

Во-вторых, логика вашего шага загрузки становится более сложной. Вам нужно подумать о том, как вы будете динамически обрабатывать изменения структуры. Будете ли вы динамически добавлять новые столбцы в таблицу в своем хранилище при обнаружении новых полей во время сбора данных? Как уведомлять аналитиков данных об изменении преобразования данных или исходных таблиц?

Если вы выберете подход структурирования при чтении, то вам нужно серьезно отнестись к *управлению данными* (data governance), которое включает не только каталогизацию ваших данных, но и определение стандартов и процессов, связанных с использованием данных в организации. Управление данными — это обширная тема, и она важна независимо от того, как вы собирае-

те данные. Тем не менее это тема, которая важна на техническом уровне, если вы выберете подход структурирования при чтении.

Масштабирование сложности конвейеров

Построение конвейеров данных, когда возможности исходных систем и последующих моделей данных ограничены, является достаточно непростой задачей. Даже в относительно небольших организациях поток данных быстро нарастает и возникает проблема масштабирования конвейеров, которые должны справляться с растущей нагрузкой. В этом разделе приведены некоторые советы и рекомендации по масштабированию на различных этапах конвейера.

Стандартизация сбора данных

Если говорить о сложности конвейера, то количество систем, из которых вы собираете данные, обычно менее важно, чем наличие различий между системами. Эти различия часто приводят к двум проблемам обслуживания конвейера:

- задания сбора данных должны быть написаны для обработки различных типов исходных систем (Postgres, Kafka и т. д.). Чем больше типов исходных систем вам нужно опросить, тем шире будет ваша кодовая база и тем выше трудозатраты по ее поддержке;
- задания сбора данных для одного и того же типа исходной системы трудно стандартизировать. Например, даже если вы получаете данные только из REST API, если эти API не имеют стандартизированных способов разбиения на страницы, инкрементного доступа к данным и других функций, инженеры данных могут создавать "разовые" задания сбора, которые не используют код повторно и не придерживаются общей логики, управляемой централизованно.

Большинство организаций располагают лишь ограниченным влиянием на системы, из которых получают данные. Возможно, вы вынуждены использовать в основном сторонние платформы, или внутренние системы создаются командой инженеров в другой части организационной иерархии. Оба этих варианта не

являются технической проблемой, но тем не менее каждый из них следует учитывать в рамках стратегии конвейера данных. К счастью, существуют особые приемы работы, которые помогут вам смягчить зависимость контейнеров от внешних факторов.

Начните с устранения организационных факторов. Если системы, из которых вы загружаете информацию, созданы внутри компании, но не стандартизированы должным образом, постарайтесь донести до владельцев систем и руководства компании мысль о негативном влиянии на конвейеры данных. Скорее всего, владельцы систем поддержат вас и пойдут навстречу.

Инженеры-программисты, создающие каждую систему, особенно в крупных компаниях, могут не осознавать, что они создают системы, которые немного отличаются от систем в других подразделениях организации. К счастью, программисты обычно понимают преимущества стандартизации в плане эффективности и удобства сопровождения. Налаживание партнерства с разработчиками требует терпения и правильного подхода, но это недооцененный нетехнический навык для команд, занимающихся обработкой данных.

Если вам необходимо получать данные из большого количества сторонних источников, то ваша организация, вероятно, во многих случаях предпочитает покупать инструменты, а не создавать их. Решения о сборке/покупке сложны, и организации обычно учитывают множество факторов при оценке различных поставщиков и предложений для собственных решений. Одним из факторов, который часто либо упускается из виду, либо оказывается весьма далек от идеала, является влияние на отчетность и аналитику. В таких случаях команды специалистов по обработке данных сталкиваются с проблемой получения данных из продукта, который не был хорошо приспособлен для этой задачи. Сделайте все возможное, чтобы принять участие в процессе обсуждения покупных продуктов как можно раньше, и добейтесь, чтобы ваши специалисты имели право голоса при выборе окончательного решения. Точно так же, как повышение осведомленности о внутренней стандартизации системы, определение потребностей аналитиков при работе с поставщиками — это фактор, который часто не принимают во внимание, если группа обработки данных не настоит на том, чтобы их голос был услышан.

Есть также некоторые технические приемы, которые помогут уменьшить сложность ваших заданий сбора данных:

- Стандартизируйте любой код, который только можете, и повторно используйте его. Это общеизвестная современная методика разработки программного обеспечения, но иногда ее игнорируют при создании заданий сбора данных.
- Стремитесь к сбору данных, построенному на основе конфигурации. Вы загружаете данные из нескольких БД и таблиц Postgres? Не пишите отдельное задание для каждого приема; разработайте одно задание, которое перебирает файлы конфигурации (или записи в таблице базы данных!), определяя таблицы и схемы, которые вы хотите принять.
- Создайте собственные абстракции. Если вы не можете заставить владельцев исходных систем построить стандартизированные абстракции между их системами и вашим вводом, подумайте о том, чтобы сделать это самостоятельно или в партнерстве с ними и взять на себя основную часть работы по разработке. Например, если вам необходимо извлечь данные из БД Postgres или MySQL, получите разрешение от команды поставщика данных на реализацию потокового CDC с помощью Debezium (см. *главу 4*) вместо разработки еще одного задания сбора данных.

Повторное использование логики модели данных

Сложность также может возрасти по мере продвижения по конвейеру и, в частности, во время моделирования данных на этапе преобразования (см. *главу 6*). Чем больше моделей данных создают аналитики, тем чаще они допускают следующие нарушения оптимальности:

- дублируют логику SQL-запросов, которые строят каждую модель;
- выводят модели друг из друга, создавая многочисленные зависимости между моделями.

Точно так же, как повторное использование кода идеально подходит для сбора данных (и разработки программного обеспечения в целом), оно также идеально подходит для моделирования

данных. Этот подход гарантирует существование единого источника достоверной информации и уменьшает объем кода, который необходимо изменить в случае ошибки или при изменении бизнес-логики. Компромиссом является более сложный граф зависимостей в конвейере.

На рис. 9.1 показан DAG (см. *главу 7*) с одним источником данных и четырьмя моделями, построенными с помощью параллельно выполняемых сценариев. Их можно выполнять таким образом, потому что они не зависят друг от друга.

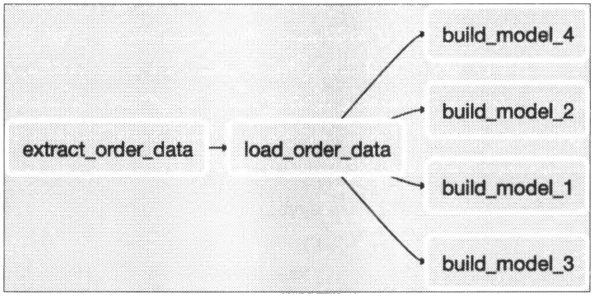

Рис. 9.1. Четыре независимые модели данных

Если модели данных действительно несвязанные, это не проблема. Однако если все они имеют какую-то общую логику, то лучше реорганизовать модели и DAG, чтобы они выглядели примерно так, как показано на рис. 9.2.

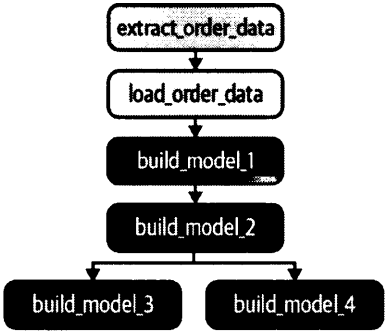

Рис. 9.2. Модели данных
с повторным использованием логики и зависимостями

В листинге 9.2 показан простой пример повторного использования логики, представляющий скрипт, выполняемый в задаче `build_model_1` на рис. 9.2. Скрипт генерирует число заказов по дням и сохраняет его в модели данных под названием `orders_by_day`.

Воспользуйтесь таблицей `Orders` из *главы 6*, которую можно воссоздать и заполнить с помощью следующего SQL-скрипта:

```
CREATE TABLE Orders (
  OrderId int,
  OrderStatus varchar(30),
  OrderDate timestamp,
  CustomerId int,
  OrderTotal numeric
);

INSERT INTO Orders VALUES(1,'Отгружен','2020-06-09',100,50.05);
INSERT INTO Orders VALUES(2,'Отгружен','2020-07-11',101,57.45);
INSERT INTO Orders VALUES(3,'Отгружен','2020-07-12',102,135.99);
INSERT INTO Orders VALUES(4,'Отгружен','2020-07-12',100,43.00);
```

Листинг 9.2. Код скрипта из файла model_1.sql

```
CREATE TABLE IF NOT EXISTS orders_by_day AS
SELECT
    CAST(OrderDate AS DATE) AS order_date,
    COUNT(*) AS order_count
FROM Orders
GROUP BY CAST(OrderDate AS DATE);
```

Последующие модели в DAG могут обращаться к этой таблице, когда им требуется ежедневное количество заказов, а не вычислять его заново. В листинге 9.3 представлен скрипт, выполняемый в задаче `build_model_2` на рис. 9.2. Вместо пересчета числа заказов по дням используется модель `orders_by_day`. Хотя подсчет числа заказов по дням может показаться тривиальным, при наличии более сложных вычислений или запросов с дополнительной логикой в операторе `WHERE` будет намного правильнее и надежнее написать логику один раз и обращаться к ней повторно. Этот подход обеспечивает единый источник достоверной информации,

единую модель для обслуживания и в качестве бонуса требует, чтобы ваше хранилище данных выполняло любую сложную логику только один раз и сохраняло результаты для последующего использования. Благодаря этому в некоторых случаях заметно сокращается время выполнения конвейера.

Листинг 9.3. Код скрипта из файла model_2.sql

```
SELECT
  obd.order_date,
  ot.order_count
FROM orders_by_day obd
LEFT JOIN other_table ot
  ON ot.some_date = obd.order_date;
```

Некоторые предусмотрительные аналитики данных с самого начала правильно проектируют свои модели данных и последующий DAG, но чаще разработчики принимаются за рефакторинг только после возникновения проблем в конвейере. Например, если в логике обнаружена ошибка, которую необходимо исправить в нескольких моделях, то, вероятно, имеет смысл оставить логику в одной модели и вывести из нее другие модели.

Хотя в итоге получается более сложный набор зависимостей, при правильном обращении, как вы увидите в следующем разделе, логика в части моделирования данных вашего конвейера становится более надежной и с меньшей вероятностью приведет к конфликту версий в разных моделях.

Обеспечение целостности зависимостей

Как отмечалось в предыдущем разделе, несмотря на все преимущества повторного использования логики модели данных, существует компромисс: необходимо отслеживать, какие модели зависят друг от друга, и обеспечивать правильное определение этих зависимостей в DAG для оркестровки.

На рис. 9.2 в предыдущем разделе (и в запросах листингов 9.2 и 9.3) `model_2` зависит от `model_1`, а `model_3` и `model_4` зависят от `model_2`. Эти зависимости должным образом определены в DAG, но по мере того, как разработчики создают больше моделей, от-

слеживание зависимостей становится довольно рутинным и подверженным ошибкам.

Поскольку мы перешли к работе с более сложными конвейерами, пришло время рассмотреть программные способы определения и проверки зависимостей между моделями данных. Существует несколько подходов, я расскажу о двух из них.

Во-первых, вы можете встроить определенную логику в свой процесс разработки, чтобы идентифицировать зависимости в сценариях SQL и гарантировать, что любые таблицы, от которых зависит сценарий, выполняются в DAG раньше. Это непростая задача, которую решают либо путем анализа имен таблиц из сценария SQL, либо, что чаще, требуя от аналитика данных, создающего модель, предоставить список зависимостей вручную в файле конфигурации при отправке новой модели или модификация существующей. В обоих случаях у вас появляется дополнительная работа.

Другой подход заключается в использовании среды разработки моделей данных, такой как dbt, которая, помимо прочих преимуществ, имеет механизм, позволяющий аналитикам определять ссылки между моделями прямо в SQL-скрипте модели.

Подробнее о dbt

dbt — это продукт с открытым исходным кодом, созданный Fishtown Analytics, который приобрел широкую популярность в сообществе аналитиков данных. Он написан на Python, его легко развернуть и использовать самостоятельно. На случай, если вы не хотите разворачивать свою среду, существует также коммерческая полностью облачная версия под названием dbt Cloud. Вы можете узнать больше о dbt, прочитав официальную документацию.

Чтобы показать, как это делается, я перепишу файл `model_2.sql` из листинга 9.3 и воспользуюсь функцией `ref()` в dbt для ссылки на `model_1.sql` при объединении. Результат показан в листинге 9.4.

Листинг 9.4. Код скрипта из файла model_2_dbt.sql

```
SELECT
  obd.order_date,
  ot.order_count
```

```
FROM {{ref('model_1')}} obd
LEFT JOIN other_table ot
  ON ot.some_date = obd.order_date;
```

Модели данных в dbt

Все модели данных в dbt определяются как операторы `SELECT`. Хотя представление моделей данных аналогично описанному в *главе 6*, в моделях dbt доступны преимущества таких функций, как `ref()`, через шаблоны Jinja, которые знакомы многим разработчикам Python.

Благодаря обновленному SQL dbt "знает", что `model_2` зависит от `model_1`, и обеспечивает выполнение в правильном порядке. На самом деле, dbt создает DAG динамически, а не заставляет вас делать это в таком инструменте, как Airflow. Когда dbt компилирует модель данных перед выполнением, ссылка на `model_1` заполняется именем таблицы (`orders_by_day`). Если все четыре модели из DAG на рис. 9.2 будут записаны в dbt, их можно скомпилировать и выполнить с помощью одной команды в командной строке:

```
$ dbt run
```

При выполнении dbt скрипты SQL, представляющие каждую модель, будут выполняться в правильном порядке в зависимости от того, как каждая таблица ссылается друг на друга. Как вы узнали из *главы 7*, запускать задачи из командной строки в Airflow очень просто. Если вы по-прежнему хотите использовать Airflow в качестве оркестратора вместе с dbt для разработки модели данных, это не проблема. На рис. 9.3 показан обновленный DAG, в котором два этапа сбора данных выполняются точно так же, как и раньше. Когда они завершены, одна задача Airflow выполняет команду запуска dbt, которая обрабатывает выполнение SQL для всех четырех моделей данных в правильном порядке.

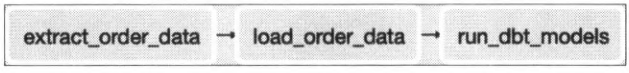

Рис. 9.3. Модели данных, выполняемые в dbt из Airflow

Хотя в этом примере я запускаю в проекте dbt все модели, вы можете указать подмножество моделей для запуска, передав параметры в `dbt run`.

Независимо от того, предпочитаете ли вы выявление и проверку зависимостей модели с помощью пользовательского кода, который вы внедряете в процесс разработки, или применяете такой продукт, как dbt, обработка зависимостей в масштабных проектах является ключом к поддержанию конвейера данных в рабочем состоянии. Старайтесь не доверять ручным проверкам и человеческим глазам!

Измерение и мониторинг производительности конвейера

Даже самые хорошо спроектированные конвейеры данных не предназначены для работы по принципу "настроил и забыл". Методика измерения и мониторинга производительности конвейеров имеет большое значение. Ваш долг перед своей командой и потребителями данных — оправдать ожидания, когда речь идет о надежности ваших конвейеров.

В этой главе изложены некоторые рекомендации относительно технологии, которую специалисты по обработке данных предоставляют другим, но, что удивительно, не всегда применяют к себе: сбор данных и измерение эффективности своей работы.

Ключевые показатели конвейера

Прежде чем начинать собирать данные со всех своих конвейеров, вы должны решить, какие показатели нужно отслеживать.

Выбор показателей должен начинаться с определения того, что важно для вас и потребителей ваших данных. Вот несколько примеров:

- сколько проверок (см. *главу 8*) выполняется и какова процентная доля успешно пройденных тестов;
- как часто конкретный DAG выполняется успешно;
- общее время работы конвейера в течение недель, месяцев и лет.

Сколько показателей отслеживать?

Остерегайтесь распространенной ловушки: просмотр слишком большого количества показателей! Хотя не следует полагаться на один показатель, чтобы рассказать всю историю производительности и надежности конвейера, избыток показателей мешает сфокусировать внимание на том,

что является наиболее важным. Я предлагаю выбрать максимум два-три показателя, на которых следует сосредоточиться. Также важно убедиться, что каждый из них уникален, а не дублирует другие измерения.

Хорошая новость заключается в том, что сбор данных, необходимых для расчета таких показателей, вполне достижим. Как вы увидите в следующих разделах, эти данные можно получать непосредственно из инфраструктуры, созданной ранее в нашей книге, в частности задействуя Airflow (*глава 7*) и фреймворк проверки данных (*глава 8*).

Подготовка хранилища данных

Прежде чем вы сможете составлять отчеты о производительности ваших конвейеров, следует собрать и сохранить данные, необходимые для такого измерения. К счастью, вы специалист по работе с данными, и все нужные инструменты уже у вас в руках! Хранилище данных — это лучшее место для размещения сведений из журналов с каждого шага конвейера данных.

В этом разделе я определяю структуру таблиц, в которых вы будете хранить данные, полученных из Airflow и фреймворка проверки данных (*глава 8*). Эти данные позже понадобятся для формирования показателей, необходимых для измерения производительности конвейера.

Я хотел бы отметить, что существует множество различных контрольных точек, которые вы можете отслеживать и включать в отчеты. Далее я выбрал два примера, потому что они хорошо иллюстрируют основы и должны вдохновить на другие действия по отслеживанию и измерениям, специфичным для вашей инфраструктуры данных.

Структура данных

Прежде всего, вам понадобится таблица для хранения истории запусков DAG из Airflow. В *главе 7* было сказано, что Airflow используется для выполнения каждого этапа в конвейере данных. Он также хранит историю каждого запуска DAG. Прежде чем извлекать эти данные, нужно приготовить таблицу для их сохранения. Далее приведено определение таблицы с именем `dag_run_`

history. Ее нужно создать в вашем хранилище в структуре, в которую вы загружаете данные во время их приема:

```
CREATE TABLE dag_run_history (
  id int,
  dag_id varchar(250),
  execution_date timestamp with time zone,
  state varchar(250),
  run_id varchar(250),
  external_trigger boolean,
  end_date timestamp with time zone,
  start_date timestamp with time zone
);
```

В дополнение к отчетам о производительности DAG важно обеспечить понимание достоверности данных. В *главе 8* был предложен простой фреймворк проверки данных на основе Python. В этой главе я дополню его, чтобы он сохранял результаты каждого проверочного теста в хранилище данных. Следующая таблица с именем validation_run_history будет местом назначения результатов проверочного теста. Я предлагаю создать ее в той же структуре вашего хранилища, куда попадают собираемые данные:

```
CREATE TABLE validation_run_history (
  script_1 varchar(255),
  script_2 varchar(255),
  comp_operator varchar(10),
  test_result varchar(20),
  test_run_at timestamp
);
```

В последующих разделах этой главы будет реализована логика для обработки данных, загруженных в две предыдущие таблицы.

Журналирование и получение данных о производительности

Теперь пришло время заполнить две таблицы, которые вы создали в своем хранилище данных с помощью SQL-запросов из предыдущего раздела. Первая будет заполнена путем создания зада-

ния сбора данных, как вы узнали из *глав 4* и *5*. Вторая потребует доработки фреймворка проверки данных, впервые представленного в *главе 8*.

Получение истории выполнения DAG из Airflow

Чтобы заполнить таблицу `dag_run_history`, только что созданную в своем хранилище данных, вам потребуется извлечь данные из БД приложения Airflow, которую вы настроили в *разделе "Настройка и знакомство с Apache Airflow" главы 7*.

В этом разделе для взаимодействия с Airflow я решил выбрать БД Postgres, поэтому следующий код извлечения следует модели, определенной в *разделе "Извлечение данных из БД PostgreSQL" главы 4*. Обратите внимание, что я выбрал инкрементную загрузку данных, что легко сделать благодаря автоинкрементному столбцу `id` таблицы `dag_run` в БД Airflow. Результатом этого извлечения (определенного в листинге 10.1) является файл CSV с именем dag_run_extract.csv. Он загружается в корзину S3, которую вы настроили в *главе 4*.

Прежде чем выполнять код, вам нужно добавить один новый раздел в файл pipeline.conf из *главы 4*. Как показано далее, он должен содержать сведения о подключении к БД Airflow, которую вы настроили в *главе 7*:

```
[airflowdb_config]
host = localhost
port = 5432
username = airflow
password = pass1
database = airflowdb
```

REST API Airflow

Хотя я получаю историю запусков DAG непосредственно из БД приложения Airflow, в идеале следует делать это через API или другой уровень абстракции. В Airflow версии 1.*x* есть "экспериментальный" REST API, который довольно ограничен и не содержит конечную точку, поддерживающую уровень детализации, необходимый для отчетов о производительности конвейера. Однако с появлением на горизонте Airflow 2.0 разработчики обещают расширенный и стабильный REST API. Я предлагаю следить за эволюцией API Airflow и в будущем рассматривать возможность загрузки из него, а не из БД приложения.

Листинг 10.1. Код скрипта airflow_extract.py

```
import csv
import boto3
import configparser
import psycopg2

# Получение учетных данных подключения Redshift
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
dbname = parser.get("aws_creds", "database")
user = parser.get("aws_creds", "username")
password = parser.get("aws_creds", "password")
host = parser.get("aws_creds", "host")
port = parser.get("aws_creds", "port")

# Подключение к кластеру Redshift
rs_conn = psycopg2.connect(
            "dbname=" + dbname
            + " user=" + user
            + " password=" + password
            + " host=" + host
            + " port=" + port)

rs_sql = """SELECT COALESCE(MAX(id),-1)
            FROM dag_run_history;"""

rs_cursor = rs_conn.cursor()
rs_cursor.execute(rs_sql)
result = rs_cursor.fetchone()

# Возвращаем только один столбец и одну строку
last_id = result[0]
rs_cursor.close()
rs_conn.commit()

# Подключение к базе данных Airflow
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
dbname = parser.get("airflowdb_config", "database")
user = parser.get("airflowdb_config", "username")
```

```
password = parser.get("airflowdb_config", "password")
host = parser.get("airflowdb_config", "host")
port = parser.get("airflowdb_config", "port")
conn = psycopg2.connect(
        "dbname=" + dbname
        + " user=" + user
        + " password=" + password
        + " host=" + host
        + " port=" + port)

# Получаем любые запуски DAG. Игнорируем работающие DAG
m_query = """SELECT
                id,
                dag_id,
                execution_date,
                state,
                run_id,
                external_trigger,
                end_date,
                start_date
            FROM dag_run
            WHERE id > %s
            AND state <> \'running\';
            """

m_cursor = conn.cursor()
m_cursor.execute(m_query, (last_id,))
results = m_cursor.fetchall()

local_filename = "dag_run_extract.csv"
with open(local_filename, 'w') as fp:
    csv_w = csv.writer(fp, delimiter='|')
    csv_w.writerows(results)

fp.close()
m_cursor.close()
conn.close()

# Загрузка значений aws_boto_credentials
parser = configparser.ConfigParser()
```

```
parser.read("pipeline.conf")
access_key = parser.get("aws_boto_credentials", "access_key")
secret_key = parser.get("aws_boto_credentials", "secret_key")
bucket_name = parser.get("aws_boto_credentials", "bucket_name")

# Загрузка CSV в корзину S3
s3 = boto3.client(
        's3',
        aws_access_key_id=access_key,
        aws_secret_access_key=secret_key)
s3_file = local_filename
s3.upload_file(local_filename, bucket_name, s3_file)
```

Завершив извлечение, вы можете загрузить содержимое CSV-файла в свое хранилище данных, как подробно описано в *главе 5*. В листинге 10.2 показано, как это сделать, если у вас есть хранилище данных Redshift.

Листинг 10.2. Код скрипта из файла airflow_load.py

```
import boto3
import configparser
import pyscopg2

# Получение учетных данных подключения Redshift
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
dbname = parser.get("aws_creds", "database")
user = parser.get("aws_creds", "username")
password = parser.get("aws_creds", "password")
host = parser.get("aws_creds", "host")
port = parser.get("aws_creds", "port")

# Подключение к кластеру Redshift
rs_conn = psycopg2.connect(
            "dbname=" + dbname
            + " user=" + user
            + " password=" + password
            + " host=" + host
            + " port=" + port)
```

```
# Загрузка account_id и iam_role из конфигурации
parser = configparser.ConfigParser()
parser.read("pipeline.conf")
account_id = parser.get(
                "aws_boto_credentials",
                "account_id")
iam_role = parser.get("aws_creds", "iam_role")

# Запуск команды COPY для сбора данных в Redshift
file_path = "s3://bucket-name/dag_run_extract.csv"

sql = """COPY dag_run_history
        (id,dag_id,execution_date,
        state,run_id,external_trigger,
        end_date,start_date)"""
sql = sql + " from %s "
sql = sql + " iam_role 'arn:aws:iam::%s:role/%s';"

# Создаем объект cursor и выполняем команду COPY
cur = rs_conn.cursor()
cur.execute(sql,(file_path, account_id, iam_role))

# Закрываем объект cursor и завершаем транзакцию
cur.close()
rs_conn.commit()

# Закрываем соединение
rs_conn.close()
```

Вы можете один раз запустить сбор данных вручную, но позже будет удобнее запланировать его с помощью DAG Airflow, как я опишу в следующем разделе этой главы.

Добавление журналирования в инструмент проверки данных

Для регистрации результатов проверочных тестов, рассмотренных в *главе 8*, я добавлю в сценарий validator.py функцию `log_result`. Поскольку скрипт уже подключается к хранилищу данных для запуска проверочных тестов, я повторно использую

это подключение и просто вставляю запись с результатом теста при помощи команды INSERT:

```
def log_result(
      db_conn,
      script_1,
      script_2,
      comp_operator,
      result):

    m_query = """INSERT INTO
                  validation_run_history(
                    script_1,
                    script_2,
                    comp_operator,
                    test_result,
                    test_run_at)
                VALUES(%s, %s, %s, %s,
                       current_timestamp);"""
    m_cursor = db_conn.cursor()
    m_cursor.execute(
      m_query,
      (script_1, script_2, comp_operator, result))
    db_conn.commit()

    m_cursor.close()
    db_conn.close()

    return
```

Для завершения модификации скрипта вам нужно будет вызвать новую функцию после запуска теста. В листинге 10.3 приведено полное определение обновленного валидатора после добавления кода журналирования. Благодаря этому дополнению каждый раз, когда запускается проверочный тест, результат записывается в таблицу validation_run_history.

Я предлагаю запустить несколько проверочных тестов, чтобы сгенерировать тестовые данные для следующих примеров. Дополнительные сведения о проведении проверочных тестов вы найдете в *главе 8*.

Листинг 10.3. Обновленный скрипт validator_logging.py

```
import sys
import psycopg2
import configparser

def connect_to_warehouse():
    # Извлечение параметров подключения из файла .conf
    parser = configparser.ConfigParser()
    parser.read("pipeline.conf")
    dbname = parser.get("aws_creds", "database")
    user = parser.get("aws_creds", "username")
    password = parser.get("aws_creds", "password")
    host = parser.get("aws_creds", "host")
    port = parser.get("aws_creds", "port")

    # Подключение к кластеру Redshift
    rs_conn = psycopg2.connect(
                "dbname=" + dbname
                + " user=" + user
                + " password=" + password
                + " host=" + host
                + " port=" + port)
    return rs_conn

# Выполнение теста, состоящего из двух скриптов
# и оператора сравнения
# Возвращаем true/false, если тест пройден/провален
def execute_test(
        db_conn,
        script_1,
        script_2,
        comp_operator):

    # Выполняем 1-й скрипт и сохраняем результат
    cursor = db_conn.cursor()
    sql_file = open(script_1, 'r')
    cursor.execute(sql_file.read())

    record = cursor.fetchone()
    result_1 = record[0]
```

```
    db_conn.commit()
    cursor.close()

    # Выполняем 2-й скрипт и сохраняем результат
    cursor = db_conn.cursor()
    sql_file = open(script_2, 'r')
    cursor.execute(sql_file.read())

    record = cursor.fetchone()
    result_2 = record[0]
    db_conn.commit()
    cursor.close()

    print("Результат 1 = " + str(result_1))
    print("Результат 2 = " + str(result_2))

    # Сравнение значений согласно comp_operator
    if comp_operator == "equals":
        return result_1 == result_2
    elif comp_operator == "greater_equals":
        return result_1 >= result_2
    elif comp_operator == "greater":
        return result_1 > result_2
    elif comp_operator == "less_equals":
        return result_1 <= result_2
    elif comp_operator == "less":
        return result_1 < result_2
    elif comp_operator == "not_equal":
        return result_1 != result_2

    # Если мы оказались здесь, что-то пошло не так
    return False

def log_result(
        db_conn,
        script_1,
        script_2,
        comp_operator,
        result):
    m_query = """INSERT INTO
                validation_run_history(
                  script_1,
```

```
                    script_2,
                    comp_operator,
                    test_result,
                    test_run_at)
                VALUES(%s, %s, %s, %s, current_timestamp);"""

    m_cursor = db_conn.cursor()
    m_cursor.execute(
                m_query,
                (script_1,
                    script_2,
                    comp_operator,
                    result)
            )

    db_conn.commit()

    m_cursor.close()
    db_conn.close()

    return

if __name__ == "__main__":
    if len(sys.argv) == 2 and sys.argv[1] == "-h":
        print("Использование python validator.py"
            + "script1.sql script2.sql "
            + "comparison_operator")
        print("Допустимые значения comparison_operator:")
        print("equals")
        print("greater_equals")
        print("greater")
        print("less_equals")
        print("less")
        print("not_equal")

        exit(0)

    if len(sys.argv) != 5:
        print("Использование python validator.py"
            + "script1.sql script2.sql "
            + "comparison_operator")
        exit(-1)
```

```
script_1 = sys.argv[1]
script_2 = sys.argv[2]
comp_operator = sys.argv[3]
sev_level = sys.argv[4]

# Подключение к хранилищу данных
db_conn = connect_to_warehouse()

# Выполнение проверки
test_result = execute_test(
                db_conn,
                script_1,
                script_2,
                comp_operator)

# Запись результата в хранилище
log_result(
    db_conn,
    script_1,
    script_2,
    comp_operator,
    test_result)

print("Результат теста: " + str(test_result))

if test_result == True:
    exit(0)
else:
    if sev_level == "halt":
        exit(-1)
    else:
        exit(0)
```

Ведение объемных журналов

Хотя ваше хранилище данных — отличное место для размещения и анализа данных о производительности вашей конвейерной инфраструктуры, сохранение журналов в основном хранилище не всегда целесообразно. Если у вас ожидается большой объем данных журнала, например результаты проверочных тестов, описанных в этом разделе, стоит подумать о том, чтобы сначала направить их в инфраструктуру анализа журналов, такую как Splunk, SumoLogic или стек ELK с открытым исходным кодом

(Elasticsearch, Logstash и Kibana). Эти платформы спроектированы так, чтобы хорошо работать с большим объемом операций записи (как это обычно бывает с записями в журнале), в то время как хранилища данных, такие как Snowflake и Redshift, лучше справляются с массовым приемом данных. После того как данные журнала будут отправлены на такую платформу, вы сможете впоследствии массово загрузить их в свое хранилище данных.

Большинство платформ ведения журналов имеют инструменты для анализа и визуализации. Я считаю, что такие инструменты предпочтительнее для изолированного анализа данных журналов, а также для оперативного мониторинга и составления отчетов о системах, создающих журналы. Тем не менее я по-прежнему считаю полезным отправлять данные журналов в свое хранилище данных для дальнейшего анализа, объединять их с источниками, не входящими в журнал, и отображать показатели производительности более высокого уровня на корпоративных информационных панелях, которыми пользуются не только инженеры. К счастью, в вашей организации уже может быть развернута и запущена необходимая инфраструктура для анализа журналов. В целом платформы анализа журналов дополняют инфраструктуру анализа данных и заслуживают отдельного изучения.

Дополнительные сведения о проведении проверочных тестов представлены в *главе 8*.

Преобразование данных о производительности

Теперь, когда вы записываете ключевые события из своих конвейеров в журнал и сохраняете в своем хранилище данных, вы можете создать на этой основе отчеты о производительности конвейера. Лучший способ сделать это — построить простой конвейер данных!

Воспользуемся шаблоном ELT, описанным в *главе 3* и используемым в этой книге. Работа по созданию отдельного конвейера для отчетов о производительности всех основных конвейеров почти завершена. Шаги извлечения и загрузки (Extract & Load, EL) были рассмотрены в предыдущем разделе. Все, что вам осталось, — это шаг преобразования (Transform, T). Для рассматриваемого конвейера это означает превращение данных о запусках Airflow DAG и других действиях, которые вы выбрали для регистрации, в показатели производительности, которые вы намереваетесь измерять и за которые несете ответственность.

В следующих подразделах я определяю преобразования для создания моделей данных некоторых ключевых показателей, рассмотренных ранее в этой главе.

Коэффициент успешного выполнения DAG

Как вы помните из *главы 6*, вы должны выбрать степень детализации данных, которые хотите смоделировать. В данном случае я хочу измерить показатель успешного выполнения каждого DAG по дням. Этот уровень детализации позволяет мне измерять успех отдельных DAG или группы из нескольких DAG ежедневно, еженедельно, ежемесячно или ежегодно. Независимо от того, запускаются ли DAG один раз в день или чаще, эта модель будет отражать нужный мне показатель успешной работы. В листинге 10.4 приведен код SQL-скрипта для построения модели. Обратите внимание, что это полностью обновленная модель.

Листинг 10.4. Код SQL-скрипта из файла dag_history_daily.sql

```
CREATE TABLE IF NOT EXISTS dag_history_daily (
  execution_date DATE,
  dag_id VARCHAR(250),
  dag_state VARCHAR(250),
  runtime_seconds DECIMAL(12,4),
  dag_run_count int
);

TRUNCATE TABLE dag_history_daily;

INSERT INTO dag_history_daily
  (execution_date, dag_id, dag_state,
  runtime_seconds, dag_run_count)
SELECT
  CAST(execution_date as DATE),
  dag_id,
  state,
  SUM(EXTRACT(EPOCH FROM (end_date - start_date))),
  COUNT(*) AS dag_run_count
FROM dag_run_history
GROUP BY
  CAST(execution_date as DATE),
```

```
  dag_id,
  state;
```

Обратившись к таблице `dag_history_daily`, вы можете измерить показатель успешности выполнения одного или всех DAG в заданном диапазоне дат. Далее приведено несколько примеров, основанных на запусках некоторых DAG, определенных в *главе 7*, но вы увидите данные, основанные на вашей собственной истории запуска Airflow DAG. Обязательно запустите хотя бы один прием данных Airflow (рассмотренный ранее в этой главе), чтобы в таблице `dag_history_daily` появились данные.

Вот запрос, возвращающий показатель успешности запусков DAG:

```
SELECT
dag_id,
SUM(CASE WHEN dag_state = 'success' THEN 1
ELSE 0 END)
/ CAST(SUM(dag_run_count) AS DECIMAL(6,2))
AS success_rate
FROM dag_history_daily
GROUP BY dag_id;
```

Конечно, вы можете применять фильтры по заданному DAG или диапазону дат. Обратите внимание, что вы должны применить к `dag_run_count` оператор приведения `CAST AS DECIMAL`, чтобы вычислить дробное значение показателя.

Результат выполнения запроса будет выглядеть примерно так:

```
dag_id                  | success_rate
------------------------+----------------------
tutorial                | 0.83333333333333333333
elt_pipeline_sample     | 0.25000000000000000000
simple_dag              | 0.31250000000000000000
(3 rows)
```

Отслеживание времени выполнения DAG

Измеряя время выполнения DAG, часто выявляют графы, выполнение которых занимает больше времени, чем ожидалось, что создает риск устаревания данных в хранилище. Я буду использо-

вать таблицу `dag_history_daily`, созданную в предыдущем подразделе, для расчета среднего времени выполнения каждого DAG по дням.

Обратите внимание, что в следующем запросе я рассматриваю только успешные запуски DAG, но у вас иногда может возникнуть потребность сообщить о длительных запусках DAG, которые завершились неудачно (возможно, из-за тайм-аута). Также имейте в виду, что, поскольку в течение одного дня может происходить несколько запусков данного DAG, необходимо усреднить время их выполнения в запросе.

Наконец, поскольку таблица `dag_history_daily` детализирована по дате и `dag_state`, мне на самом деле нет нужды обязательно суммировать `runtime_seconds` и `dag_run_count`, но лучше все-таки это сделать. Почему? Если бы я или другой аналитик решили изменить логику, чтобы сделать что-то вроде подсчета неудачных запусков DAG, то потребовалась бы функция `SUM()`, но ее легко пропустить.

Так выглядит запрос для DAG `elt_pipeline_sample` из *главы 7*:

```
SELECT
  dag_id,
  execution_date,
  SUM(runtime_seconds)
    / SUM(CAST(dag_run_count as DECIMAL(6,2)))
  AS avg_runtime
FROM dag_history_daily
WHERE
  dag_id = 'elt_pipeline_sample'
GROUP BY
  dag_id,
  execution_date
ORDER BY
dag_id,
execution_date;
```

Результат выполнения запроса будет выглядеть примерно так:

```
dag_id               | execution_date | avg_runtime
---------------------+----------------+----------
elt_pipeline_sample  | 2020-09-16     | 63.773900
elt_pipeline_sample  | 2020-09-17     | 105.902900
```

```
elt_pipeline_sample  | 2020-09-18      | 135.392000
elt_pipeline_sample  | 2020-09-19      | 101.111700
(4 rows)
```

Объем выполненных тестов и доля успешных результатов

Благодаря дополнительному журналированию, которое вы добавили в инструмент проверки данных ранее в этой главе, теперь можно измерить долю успешно пройденных тестов, а также общий объем выполненных тестов.

Объем тестов в контексте

Внимания заслуживает как показатель успешного прохождения, так и объем выполненных тестов, хотя я предлагаю также рассматривать объем тестов в контексте системы. Подробности этого подхода немного выходят за рамки данной главы, но суть в том, что число выполненных проверок должно быть пропорционально количеству запущенных задач DAG. Другими словами, вы должны убедиться, что тестируете каждый шаг в своих конвейерах. Каково правильное соотношение тестов к этапам конвейера (часто измеряемое задачей DAG)? Это зависит от того, насколько сложны ваши этапы. Для простых этапов загрузки может потребоваться один тест для проверки повторяющихся строк, в то время как некоторые этапы преобразования заслуживают нескольких тестов для проверки различных ошибок, зависящих от контекста.

В листинге 10.5 приведена новая модель данных с именем `validator_sum mary_daily`, которая вычисляет и сохраняет результаты каждого теста с детализацией по дням.

Листинг 10.5. Код скрипта из файла validator_summary_daily.sql

```
CREATE TABLE IF NOT EXISTS validator_summary_daily (
  test_date DATE,
  script_1 varchar(255),
  script_2 varchar(255),
  comp_operator varchar(10),
  test_composite_name varchar(650),
  test_result varchar(20),
  test_count int
);
```

```
TRUNCATE TABLE validator_summary_daily;

INSERT INTO validator_summary_daily
  (test_date, script_1, script_2, comp_operator,
  test_composite_name, test_result, test_count)
SELECT
  CAST(test_run_at AS DATE) AS test_date,
  script_1,
  script_2,
  comp_operator,
  (script_1
    || ' '
    || script_2
    || ' '
    || comp_operator) AS test_composite_name,
  test_result,
  COUNT(*) AS test_count
FROM validation_run_history
GROUP BY
  CAST(test_run_at AS DATE),
  script_1,
  script_2,
  comp_operator,
  (script_1 || ' ' || script_2 || ' ' || comp_operator),
  test_result;
```

Хотя логика создания `validator_summary_daily` довольно проста, стоит отдельно упомянуть столбец `test_composite_name`. При отсутствии уникального имени для каждого проверочного теста (улучшение, о котором стоит подумать) `test_composite_name` представляет собой комбинацию двух скриптов и оператора для теста. Он действует как составной ключ, который позволяет сгруппировать прогоны теста. В качестве примера далее показан SQL-скрипт для расчета процентной доли времени прохождения каждого теста. Разумеется, вы можете просмотреть статистику по дням, неделям, месяцам или любому другому временному диапазону:

```
SELECT
  test_composite_name,
```

```
  SUM(
    CASE WHEN test_result = 'true' THEN 1
    ELSE 0 END)
    / CAST(SUM(test_count) AS DECIMAL(6,2))
  AS success_rate
  FROM validator_summary_daily
  GROUP BY
    test_composite_name;
```

Результат выполнения запроса будет выглядеть примерно так:

```
test_composite_name         | success_rate
----------------------------+---------------------
sql1.sql sql2.sql equals    | 0.33333333333333333
sql3.sql sql4.sql not_equal | 0.75000000000000000
(2 rows)
```

Что касается объема проверок, вы можете получить отчет по заданной дате, по результатам определенного теста или по обоим критериям одновременно. Как отмечалось ранее, важно рассматривать этот показатель в контексте системы. По мере увеличения количества и сложности конвейеров регулярно проверяйте их, чтобы убедиться, что у вас нет пробелов проверки достоверности данных во всех конвейерах. Следующий SQL-запрос выдает как число тестов, так и процент успешных результатов по дате. Это набор данных, который вы можете нанести на линейную диаграмму с двойной осью Y или другой график:

```
SELECT
  test_date,
  SUM(
    CASE WHEN test_result = 'true' THEN 1
    ELSE 0 END)
    / CAST(SUM(test_count) AS DECIMAL(6,2))
  AS success_rate,
  SUM(test_count) AS total_tests
  FROM validator_summary_daily
  GROUP BY
    test_date
  ORDER BY
    test_date;
```

Результаты выполнения запроса будут выглядеть примерно так:

```
test_date   | success_rate | total_tests
------------+--------------------+-----------
2020-11-03  | 0.3333333333333333333 | 3
2020-11-04  | 1.0000000000000000000 | 6
2020-11-05  | 0.5000000000000000000 | 8
(3 row)
```

Оркестровка конвейера производительности

На основе примеров кода из предыдущих разделов вы можете создать новый DAG Airflow для планирования и оркестровки конвейера, занятого сбором и преобразованием данных о производительности ваших производственных конвейеров. Этот подход может показаться немного рекурсивным, но зато у вас будет готовая инфраструктура для этого типа операций. Только не забывайте, что эти ретроспективные отчеты предназначены для отображения аналитических данных, а не критически важной информации, такой как мониторинг времени безотказной работы или оповещение о конвейерах. Вы никогда не должны использовать одну и ту же инфраструктуру для рабочего процесса и нужд оперативного мониторинга!

DAG конвейера производительности

DAG для оркестровки всех шагов, описанных в этой главе, будет похож на примеры из *главы 7*. Благодаря коду из листинга 10.3 результаты проверочных тестов уже регистрируются в хранилище данных. Это означает, что в конвейере мониторинга производительности нужно реализовать всего несколько шагов:

1. Извлечение данных из БД Airflow (листинг 10.1).
2. Загрузка данных в хранилище (листинг 10.2).
3. Преобразование истории Airflow (листинг 10.4).
4. Преобразование истории проверки данных (листинг 10.5).
5. В листинге 10.6 приведен код DAG Airflow, а на рис. 10.1 изображена схема DAG.

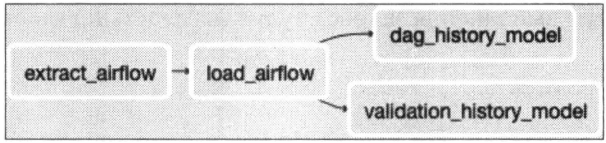

Рис. 10.1. Графическое представление DAG pipeline_performance

Листинг 10.6. Код скрипта pipeline_performance.py

```
from datetime import timedelta
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.postgres_operator import PostgresOperator
from airflow.utils.dates import days_ago

dag = DAG(
    'pipeline_performance',
    description='Конвейер измерения производительности',
    schedule_interval=timedelta(days=1),
    start_date = days_ago(1),
)

extract_airflow_task = BashOperator(
    task_id='extract_airflow',
    bash_command='python /p/airflow_extract.py',
    dag=dag,
)

load_airlflow_task = BashOperator(
    task_id='load_airflow',
    bash_command='python /p/airflow_load.py',
    dag=dag,
)

dag_history_model_task = PostgresOperator(
    task_id='dag_history_model',
    postgres_conn_id='redshift_dw',
    sql='/sql/dag_history_daily.sql',
    dag=dag,
)
```

```
validation_history_model_task = PostgresOperator(
    task_id='validation_history_model',
    postgres_conn_id='redshift_dw',
    sql='/sql/validator_summary_daily.sql',
    dag=dag,
)

extract_airflow_task >> load_airlflow_task
load_airlflow_task >> dag_history_model_task
load_airlflow_task >> validation_history_model_task
```

Раскрытие информации о производительности

Обладателю отдельного конвейера для измерения производительности производственных конвейеров и тестов проверки данных нужно придерживаться простого правила: делиться полученными знаниями со своей командой и всеми заинтересованными сторонами. Прозрачность работы конвейера — это ключ к построению доверительных отношений с пользователями и коллегами и поддержанию командного духа.

Вот несколько советов по использованию данных и выводов, полученных в этой главе:

- Применяйте инструменты визуализации. Сделайте метрики созданных вами моделей данных доступными в тех же инструментах визуализации, которые используют ваши потребители данных или владельцы системы. Это может быть Tableau, Looker или аналогичный продукт. Что бы это ни было, сделайте так, чтобы данные о производительности отображались там, куда члены вашей команды и потребители данных заглядывают каждый день.
- Регулярно (хотя бы раз в месяц, если не раз в неделю) делитесь сводными показателями по электронной почте, в Slack или в другом канале для обмена рабочей информацией.
- Следите за тенденциями, а не только за текущими значениями. Как на информационных панелях, так и в сводках не просто делитесь последними значениями каждой метрики. Отслежи-

вайте также изменения с течением времени и убедитесь, что отрицательные тенденции отображаются так же часто, как и положительные.

- Реагируйте на тенденции. Отображение тенденций на информационных панелях нужно не просто для "галочки". Это возможность вовремя отреагировать и исправить неполадки. Вы заметили, что проверки данных проваливаются чаще, чем месяцем ранее? Выясните причину, внесите изменения и продолжайте следить за тенденциями, чтобы оценить результат своей работы.

Предметный указатель

П

С

Т

Об авторе

Джеймс Денсмор — директор по инфраструктуре данных в HubSpot, а также основатель и главный консультант Data Liftoff. Он обладает более чем десятилетним опытом руководства группами по работе с данными и создания инфраструктуры данных в Wayfair, O'Reilly Media, HubSpot и Degreed. Джеймс имеет степень бакалавра компьютерных наук Северо-Восточного университета и степень магистра делового администрирования Бостонского колледжа.

Об изображении на обложке

Птица на обложке справочника по конвейерам данных — это белобровая шилоклювая тимелия (Pomatostomus superciliosus). Слово superciliosus происходит от латинского supercilium, или бровь, что указывает на главную отличительную черту птицы.

У белобровых тимелий белые линии бровей и горло. Цвет их оперения варьируется от серо-коричневого до темно-коричневого. Это самые маленькие из австралийских тимелий длиной от 15 до 20 сантиметров, с длинными хвостами и короткими крыльями. Эти птицы проводят большую часть своего времени в лесах южной Австралии в поисках насекомых, ракообразных, фруктов, семян и орехов.

В отличие от большинства птиц, белобровые тимелии строят два гнезда: одно для ночлега, другое — для высиживания яиц. Птицы очень общительны: собираясь в большие группы, они шумят так сильно, что австралийцы также называют их болтунами, гоготуньями или горлопанами.

Белобровые тимелии имеют статус вида наименьшего риска, присвоенный МСОП. Многие животные на обложках O'Reilly находятся под угрозой исчезновения; все они важны для мира.

Иллюстрация на обложке выполнена Карен Монтгомери на основе черно-белой гравюры Encyclopedie D'Histoire Naturelle.

Джеймс Денсмор

Конвейеры данных. Карманный справочник

Перевод с английского

ТОО "АЛИСТ"
010000, Республика Казахстан,
г. Астана, пр. Сарыарка, д. 17, ВП 30

Подписано в печать 06.03.24.
Формат 60×90 1/16. Печать офсетная. Усл. печ. л. 16.
Тираж 1200 экз. Заказ № 8888.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, РФ, М.О., г. Чехов, ул. Полиграфистов, д. 1