

**ВЫСШЕЕ
ОБРАЗОВАНИЕ**

А. В. Маркин

Программирование на SQL Часть 2

Учебник и практикум
2-е издание

УМО ВО
РЕКОМЕНДУЕТ

 **Юрайт**
ПРЕДПРИЯТИЕ
biblio-online.ru

А. В. Маркин

ПРОГРАММИРОВАНИЕ НА SQL ЧАСТЬ 2

УЧЕБНИК И ПРАКТИКУМ ДЛЯ ВУЗОВ

2-е издание, исправленное и дополненное

*Рекомендовано Учебно-методическим отделом высшего образования в качестве
учебника и практикума для студентов высших учебных заведений, обучающихся
по инженерно-техническим направлениям*

**Книга доступна на образовательной платформе «Юрайт» urait.ru,
а также в мобильном приложении «Юрайт.Библиотека»**

Москва • Юрайт • 2022

УДК 004.42(075.8)
ББК ББК 32.973-018я73
М26

Автор:

Маркин Александр Васильевич — кандидат технических наук, старший научный сотрудник, доцент, почетный работник высшего профессионального образования Российской Федерации, доцент кафедры автоматизированных систем управления факультета автоматики и информационных технологий в управлении Рязанского государственного радиотехнического университета.

Рецензенты:

Коваленко В. В. — кандидат технических наук, доцент кафедры информационных технологий Сочинского государственного университета;

Белов В. В. — доктор технических наук, профессор кафедры вычислительной и прикладной математики Рязанского государственного радиотехнического университета.

Маркин, А. В.

М26 Программирование на SQL. В 2 частях. Ч. 2 : учебник и практикум для вузов / А. В. Маркин. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2022. — 340 с. — (Высшее образование). — Текст : непосредственный.

ISBN 978-5-534-12258-9 (ч. 2)

ISBN 978-5-534-12257-2

Подробно рассмотрены основные синтаксические конструкции языка структурированных запросов SQL, применяемые при декларативном и процедурном программировании для современных систем управления реляционными базами данных Firebird, Microsoft SQL Server и Oracle Database. Теоретический материал в достаточном объеме проиллюстрирован примерами запросов и скриптов для учебной базы данных, являющейся упрощенной моделью базы данных реальной информационной системы. Разработанные автором дополнительные интернет-ресурсы позволяют читателю самостоятельно приобрести практические навыки программирования на SQL, а также проверить полученные теоретические знания, выполнив оригинальный лабораторный практикум и пройдя тестирование по всем разделам учебника.

Содержание учебника соответствует актуальным требованиям федерального государственного образовательного стандарта высшего образования.

Для студентов высших учебных заведений, обучающихся по направлениям, связанным с разработкой, эксплуатацией, администрированием и сопровождением реляционных баз данных информационных систем, а также для всех желающих самостоятельно изучить основы SQL.

УДК 004.42(075.8)
ББК ББК 32.973-018я73

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

ISBN 978-5-534-12258-9 (ч. 2)
ISBN 978-5-534-12257-2

© Маркин А. В., 2017
© Маркин А. В., 2019, с изменениями
© ООО «Издательство Юрайт», 2022

Оглавление

Глава 5. Программирование запросов модификации данных	5
5.1. Добавление данных	6
5.1.1. Однострочный запрос INSERT	10
5.1.2. Запрос INSERT с подзапросом.....	12
5.2. Обновление данных	16
5.2.1. Явный запрос UPDATE	19
5.2.2. Запрос UPDATE с подзапросом	22
5.3. Удаление данных	31
5.3.1. Явный запрос DELETE	33
5.3.2. Запрос DELETE с подзапросом.....	33
5.3.3. Удаление всех строк таблицы	39
5.4. Условное манипулирование данными.....	40
5.4.1. Запрос UPDATE OR INSERT	40
5.4.2. Запрос MERGE.....	43
5.5. Обновление представлений	48
<i>Контрольные вопросы</i>	59
Глава 6. Процедурное программирование.....	60
6.1. Основы разработки исполняемых модулей	62
6.1.1. SQL-сценарии	63
6.1.2. Переменные	77
6.1.3. Операторы ветвления, циклов и переходов	84
6.1.4. Курсоры	90
6.1.5. Последовательности	102
6.1.6. Обработка исключительных ситуаций	109
6.2. Хранимые процедуры.....	116
6.2.1. Определение, модификация и удаление процедур	117
6.2.2. Виды хранимых процедур.....	123
6.2.3. Селективные процедуры Firebird	128
6.2.4. Выполняемые процедуры	139
6.2.5. Динамический SQL	167
6.3. Хранимые функции	176
6.4. Процедурные пакеты.....	181
6.5. Выполнимые блоки	187
6.6. Триггеры.....	189
6.6.1. Триггеры DML	192
6.6.2. Триггеры DDL.....	225

6.6.3. Триггеры баз данных	236
<i>Контрольные вопросы</i>	239
Глава 7. Программирование запросов управления доступом ...	241
7.1. Принципы доступа к данным. Управление пользователями	242
7.2. Схемы базы данных в MS SQL Server	249
7.3. Привилегии доступа и передача привилегий	251
7.4. SQL-роли	262
7.5. Привилегии на представления	268
7.6. Удаление привилегий	269
7.7. Отмена привилегий	273
7.8. Автоматизированное управление доступом	274
<i>Контрольные вопросы</i>	277
Глава 8. Управление транзакциями	278
8.1. Понятие транзакции	279
8.2. Восстановление данных	280
8.3. Параллелизм	286
<i>Контрольные вопросы и задания</i>	291
Лабораторный практикум	292
Лабораторная работа № 1. IBExpert — инструмент разработчика и администратора баз данных Firebird. Учебная база данных	292
Лабораторная работа № 2. Программирование однотоабличных запросов выборки данных	296
Лабораторная работа № 3. Программирование многотоабличных и вложенных запросов выборки данных	297
Лабораторная работа № 4. Программирование запросов определения данных	298
Лабораторная работа № 5. Программирование запросов модификации данных	300
Лабораторная работа № 6. Программирование хранимых процедур	304
Лабораторная работа № 7. Программирование триггеров	310
Лабораторная работа № 8. Программирование запросов управления доступом	313
Приложение А. Описание учебной базы данных	316
Приложение Б. Системный каталог базы данных Firebird	328
Б.1. Общие сведения о базе данных	328
Б.2. Домены	329
Б.3. Отношения (таблицы и представления)	330
Б.4. Процедуры	332
Б.5. Триггеры	333
Б.6. Зависимости	334
Библиографический список	336
Новинки по дисциплине	339

Глава 5

ПРОГРАММИРОВАНИЕ ЗАПРОСОВ МОДИФИКАЦИИ ДАННЫХ

В гл. 4 (часть 1) были изучены возможности SQL по определению структур таблиц и представлений БД. В настоящей главе рассматривается, каким образом осуществляется манипулирование данными в БД.

Изменение данных в БД выполняется с помощью запросов языка манипулирования данными — DML (Data Manipulation Language). В SQL СУБД Firebird существуют следующие запросы на изменение данных:

- INSERT (вставка данных);
- UPDATE (обновление существующих данных);
- UPDATE OR INSERT (обновление или вставка данных);
- MERGE (обновление или вставка данных таблицы на основе строк, отобранных из другой таблицы);
- DELETE (удаление данных).

Запросы DML иногда называют запросами действия (*action queries*), поскольку они изменяют информацию, записанную в таблицах [1].

К составлению таких запросов следует относиться внимательно, так как при неправильной формулировке внесенные в БД изменения могут нанести непоправимый урон и привести к потере данных. Выполнение запросов DML может привести к нарушению целостности и корректности данных, к потере их достоверности и непротиворечивости.

Чтобы информация, хранящаяся в БД, оставалась однозначной и непротиворечивой, при внесении изменений в данные объектов БД следует учитывать правила целостности, которые, по сути, являются ограничениями для всех допустимых состояний БД и гарантируют корректность данных. Ограничения целостности позволяют свести к минимуму ошибки, возникающие при добавлении, обновлении и удалении данных.

Использование вышеуказанных запросов по отношению к базовым таблицам является предметом данной главы. Также здесь приводится описание правил манипулирования данными обновляемых представлений.

5.1. Добавление данных

Добавление данных в определенную таблицу или представление БД осуществляется вставкой одной или нескольких строк с помощью запроса-действия INSERT. Существует два вида этого запроса: однострочный и многострочный. Независимо от того, какая СУБД используется, запрос INSERT можно представить в общем виде. Указываются таблица, список столбцов и список добавляемых в эти столбцы значений. Однострочный запрос выглядит так:

```
INSERT INTO <таблица> [(<список столбцов>)]  
VALUES (<список значений>);
```

а многострочный запрос содержит подзапрос SELECT:

```
INSERT INTO <таблица> [(<список столбцов>)]  
(SELECT <список значений> FROM <откуда>  
WHERE <список условий>);
```

В случае многострочного запроса в указанную таблицу (в перечисленные через запятую столбцы) выполняется копирование выбранного с помощью запроса SELECT набора данных. Естественно, что количество и тип столбцов в запросе SELECT должны соответствовать количеству и типу столбцов таблицы, в которую осуществляется вставка.

В Firebird запрос INSERT можно представить в обобщенном виде:

```
INSERT INTO { базовая_таблица | представление }  
[(столбец1 [, столбец2] ...)]  
{VALUES ({значение1 | переменная1 | NULL  
| (<скалярный_подзапрос1>)}  
[, {значение2 | переменная2 | NULL | (<скалярный_подзапрос2>)}]...)  
| <запрос_SELECT>  
| DEFAULT VALUES }  
[RETURNING <список_столбцов> [INTO <список_переменных>]]};
```

где

<значение> ::= { столбец | константа | <выражение> | функция }

Секция INTO определяет таблицу или представление и столбцы, в которые будут вставлены строки.

В секции VALUES задаются значения, которые будут присвоены столбцам добавляемых строк. В качестве значения могут использоваться и контекстные переменные (например, CURRENT_USER в контексте текущего соединения клиента). Также можно установить для столбца NULL-маркер. При однократном использовании однострочного запроса INSERT в таблицу или представление можно

вставить только одну строку. Если применить запрос INSERT с вложенным запросом SELECT (многострочный запрос INSERT), то с помощью одного запроса можно вставить в таблицу или представление сразу несколько строк из другой таблицы. Секция DEFAULT VALUES позволяет вставлять строки вообще без указания значения — ни напрямую, ни через запрос SELECT. Это возможно только, если каждый столбец таблицы с проверкой NOT NULL или CHECK имеет объявленное значение по умолчанию или получает значение в триггере до вставки (BEFORE INSERT).

Секция RETURNING позволяет запоминать значения необходимых столбцов в изменяемой строке. Ее использование во всех запросах модификации данных (INSERT, UPDATE OR INSERT, UPDATE, DELETE, MERGE) будет рассмотрено при изучении процедурного языка (подпараграф 6.2.4).

Реализация запроса INSERT в Oracle позволяет вставлять данные в указанную таблицу, представление, раздел, подраздел или объектную таблицу, а также поддерживаются дополнительные расширения, такие, как вставка строк сразу в несколько таблиц и вставка по условию [2, 3]. Синтаксис стандартного запроса имеет вид

```
INSERT [INTO] <целевой_объект> [псевдоним] [(столбец1
                                     [, столбец2] ...)]
{VALUES ( {выражение1 | DEFAULT} [, {выражение2 | DEFAULT}] ...)
  | <запрос_SELECT>}
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

где

```
<целевой_объект> ::=
{ таблица [[SUB]PARTITION (имя_раздела)]
  | представление
  | (<запрос_SELECT> [WITH {READ ONLY | CHECK OPTION
[CONSTRAINT имя_ограничения]]])
  | TABLE (коллекция) [(+)] }
```

Синтаксис запроса INSERT с условием:

```
INSERT {[ALL | FIRST]}
WHEN <условие1> THEN INTO <целевой_объект1> [псевдоним]
[(столбец1 [, столбец2] ...)]
  VALUES ( {выражение1 | DEFAULT} [, {выражение2 | DEFAULT}] ...)
[WHEN <условие2> THEN INTO <целевой_объект2> [псевдоним]
[(столбец1 [, столбец2] ...)]
  VALUES ( {выражение1 | DEFAULT} [, {выражение2 | DEFAULT}] ...) ]...
ELSE INTO <целевой_объект3> [псевдоним] [(столбец1 [, столбец2] ...)]
  VALUES ( {выражение1 | DEFAULT} [, {выражение2 | DEFAULT}] ...)
<запрос_SELECT>;
```

Секция INTO определяет целевой объект, его псевдоним и столбцы, куда будут вставлены строки. Если строки вставляются в секционированную таблицу Oracle (см. подп. 2.5.4), то может быть указан раздел или подраздел в конструкции PARTITION или SUBPARTITION соответственно.

Один из способов реализации многотабличной вставки в Oracle — указание в секции INTO подзапроса SELECT. Все столбцы, определяемые в подзапросе, во всех его таблицах, должны иметь соответствующее значение для вставки, в противном случае возникнет ошибка. По сути, с помощью подзапроса создается представление «на ходу», и запрос вставки данных в подзапрос аналогичен запросу вставки данных в представление. Опция WITH READ ONLY показывает, что таблицу, к которой обращается подзапрос, нельзя обновлять до его завершения. Опция WITH CHECK OPTION [CONSTRAINT имя_ограничения] показывает, что в таблицу или представление нельзя вставлять строки, не удовлетворяющие указанному проверочному (CHECK) ограничению.

Платформа Oracle позволяет работать с объектными таблицами, которые допускают хранение неатомарных (нескалярных) значений в полях, а именно объекта в смысле объектного подхода. Для вставки данных в такие таблицы используется конструкция TABLE (коллекция) [(+)].

Поскольку определение целевого объекта для вставки данных в секции INTO может быть очень длинным, есть возможность дополнительно указывать псевдонимы. Псевдонимы нельзя использовать при многотабличной вставке. Также при многотабличной вставке не используется секция RETURNING.

Секция VALUES предполагает определение вставляемых значений в виде выражения, которое может включать константы, переменные либо представлять собой скалярный подзапрос SELECT, или в виде DEFAULT (значения по умолчанию для столбца).

Реализация запроса INSERT с условием позволяет выполнять в Oracle операции многотабличной вставки данных (в несколько таблиц) по заданному условию. Опция ALL применяется только с подзапросами. Если не указана секция WHEN, то все данные, возвращаемые подзапросом, без всяких условий вставляются в указанные таблицы. При использовании опции ALL с секцией WHEN выполняются условные операции вставки, и все секции WHEN проверяются независимо от результатов других операций WHEN. Если условие WHEN возвращает значение true, то Oracle выполняет соответствующую секцию INTO. Многотабличную вставку по условию нельзя сделать параллельной в таблицах с индексом на основе битовых карт. Такая вставка вообще не разрешена в следующих ситуациях:

- целью является представление или материализованное представление;
- целью является таблица на удаленном сервере;

СУБД при задании имени связанного сервера или при использовании функций OPENQUERY и OPENROWSET. При использовании запроса INSERT в процедурном языке в качестве таблицы может быть указана заранее объявленная табличная переменная (подп. 6.1.2).

Необязательная секция OUTPUT может использоваться для возврата данных всех строк, изменившихся в результате выполнения запроса. Позволяет поместить все добавленные запросом строки либо в локальную переменную с типом данных TABLE либо в указанную таблицу БД (обычную или временную).

Сами значения, помещаемые в столбцы новой строки таблицы, могут быть заданы одним из четырех способов:

- 1) в секции VALUES с перечислением значений;
- 2) с помощью запроса SELECT, который возвращает одну или более строк таблицы;
- 3) при вызове хранимой процедуры (ХП) оператором EXECUTE;
- 4) присваиванием значений по умолчанию всем столбцам (DEFAULT VALUES).

В случае задания секции VALUES можно в одном запросе добавлять множество строк, повторяя через запятую список значений в скобках.

5.1.1. Однострочный запрос INSERT

Формат однострочного запроса-действия INSERT в Firebird имеет вид

```
INSERT INTO {базовая_таблица | представление} [(столбец1
                                                [, столбец2] ...)]
{VALUES ({значение1 | переменная1 | NULL
        | (<скалярный_подзапрос1>)} [, {значение2
        | переменная2 | NULL
        | (<скалярный_подзапрос2>)}] ... )
| DEFAULT VALUES }
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

В таблицу или представление вставляется строка со значениями столбцов, указанными в перечне секции VALUES, причем i -е значение соответствует i -му столбцу в списке столбцов. Столбцы, не указанные в списке, заполняются значениями по умолчанию или NULL, если ограничение, наложенное на столбец, позволяет вставку NULL. Если допустимы NULL-маркеры и определено значение по умолчанию, то вставлено будет значение по умолчанию. Если NULL недопустимы и не определено значение по умолчанию, то запрос на вставку без явного указания значения такого столбца выполнен не будет.

Примечание. Если NULL недопустимы, то можно ввести, например, пустую строку для символьного столбца, однако такую подмену делать не рекомендуется.

В Oracle формат однострочного запроса INSERT имеет вид

```
INSERT [INTO] <целевой_объект> [псевдоним] [(столбец1 [, столбец2]
...)]
VALUES ( {выражение1 | DEFAULT} [{,}{выражение2 | DEFAULT}]...)
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

Microsoft SQL Server поддерживает однострочный запрос INSERT в формате

```
INSERT [ INTO ] { [[<сервер>].][[<база данных>].<схема>.]
{ <таблица> | <представление> }
| { <функция OPENQUERY> | <функция OPENROWSET> } }
[ WITH (<подсказка таблицы> [, <подсказка таблицы>...]) ]
[ (<список столбцов>) ] [ <секция_OUTPUT> ]
{ VALUES ( { DEFAULT | NULL | <выражение1> } [, { DEFAULT | NULL |
<выражение2> } ]...)
| <оператор_EXECUTE>
| DEFAULT VALUES};
```

Рассмотрим вставку новой строки в таблицу при полном указании списка ее столбцов. Например, для вставки строки в таблицу Abonent можно использовать запрос

```
INSERT INTO Abonent (AccountCD, StreetCD,
                    HouseNo, FlatNo, Fio, Phone)
VALUES ('500000', 8, 1, 1, 'Плитов Е. Д.', '556787');
```

Если в списке опции VALUES указаны значения для всех столбцов модифицируемой таблицы и порядок их перечисления соответствует порядку столбцов в описании таблицы (как в приведенном примере), то список столбцов в секции INTO можно опустить. Например, предыдущий запрос на вставку строки в таблицу Abonent можно записать так:

```
INSERT INTO Abonent VALUES ('500000', 8, 1, 1,
                             'Плитов Е. Д. ', '556787');
```

Если требуется ввести NULL-маркер, например, в столбец Phone (номер телефона абонента неизвестен), то он вводится точно так же, как и обычное значение, например:

```
INSERT INTO Abonent VALUES ('500000', 8, 1, 1, 'Плитов Е. Д.', NULL);
```

В примерах значения, помещаемые в таблицу, располагаются в том порядке, в котором они были созданы запросом CREATE TABLE. Но в некоторых случаях требуется поменять порядок вводимых значений или вводить значения не во все столбцы в таблице. Чтобы удовлетворить этому условию, требуется явно указывать имена и порядок столбцов. Примером этому может служить запрос

```
INSERT INTO Abonent (AccountCD, StreetCD, Fio)
VALUES ('500000', 8, 'Плитов Е. Д.');
```

Использовать результат скалярного подзапроса в качестве значения можно, например, так:

```
INSERT INTO Abonent VALUES ('500000', 8, 1, 1, 'Плитов Е. Д.', (
  SELECT Phone FROM Abonent WHERE AccountCD = '005488'));
```

Все приведенные запросы INSERT корректно работают в любой из рассматриваемых в учебнике СУБД.

5.1.2. Запрос INSERT с подзапросом

Запрос-действие INSERT с подзапросом позволяет вставить в таблицу или представление сразу несколько строк.

В Firebird многострочный запрос INSERT реализуется путем задания вместо секции VALUES подзапроса SELECT, который позволяет вставить в одну таблицу или представление несколько строк из другой таблицы. Формат запроса INSERT с подзапросом SELECT в Firebird имеет вид

```
INSERT INTO { базовая_таблица | представление } [(столбец1 [, столбец2]
...)]
<запрос_SELECT>
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

В этом формате запроса INSERT сначала выполняется запрос SELECT, с помощью которого в памяти формируется рабочая таблица, а затем строки этой рабочей таблицы загружаются в модифицируемую таблицу или представление. При этом *i*-й столбец рабочей таблицы соответствует *i*-му столбцу в списке столбцов модифицируемой таблицы или представления. При использовании многострочного запроса INSERT требуется следить за тем, чтобы тип и количество значений, возвращаемых подзапросом SELECT, совпадали с типом и количеством столбцов в модифицируемой таблице.

В Oracle многострочный запрос INSERT реализуется также путем использования подзапроса SELECT:

```
INSERT [INTO] <целевой_объект> [псевдоним] [(столбец1 [, столбец2] ...)]
<запрос_SELECT>
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

Помимо этого, запрос с условием в Oracle (синтаксис был приведен в параграфе 5.1) — это также многострочный INSERT.

В MS SQL Server многострочный запрос INSERT может быть реализован не только путем использования подзапроса SELECT вместо секции VALUES, но также путем использования подзапроса как

обобщенного табличного выражения или множественного задания списка значений в секции VALUES. Формат многострочного запроса в MS SQL Server:

```
[ <предложение обобщенного табличного выражения> ]
INSERT [ TOP (<выражение>) [ PERCENT ] ]
[ INTO ] { [<сервер>.] [<база данных>.<схема>.]
{ <таблица> | <представление> }
| { <функция OPENQUERY> | <функция OPENROWSET> } }
[ WITH (<подсказка таблицы> [, <подсказка таблицы>...]) ]
[ (<список столбцов>) ] [ <секция_OUTPUT> ]
{ VALUES ( { DEFAULT | NULL | <выражение> }
            [, { DEFAULT | NULL | <выражение> } ]...),
          ( { DEFAULT | NULL | <выражение> }
            [, { DEFAULT | NULL | <выражение> } ]...)...
| <запрос_SELECT>;
```

Рассмотрим примеры использования запросов INSERT с подзапросом SELECT в Firebird.

Пусть имеется пустая таблица Fio со столбцами Abonent_Name и Executor_Name, определенными на типе данных VARCHAR(20). В эту таблицу необходимо поместить ФИО абонентов и исполнителей каждой из ремонтных заявок. Для этого можно применить запрос

```
INSERT INTO Fio (Abonent_Name, Executor_Name)
      SELECT A.Fio, E.Fio FROM Abonent A, Executor E, Request R
      WHERE R.AccountCD = A.AccountCD AND
            R.ExecutorCD = E.ExecutorCD;
```

В секции FROM запроса SELECT может быть указана не базовая таблица, а представление. Таким образом, можно вставлять данные из представления в базовую таблицу. Например, если ранее было создано представление Abonent_Executor со столбцами Abonent_Name и Executor_Name, то предыдущий запрос можно переписать:

```
INSERT INTO Fio
      SELECT Abonent_Name, Executor_Name FROM Abonent_Executor;
```

Содержание таблицы Fio после вставки будет совпадать с данными, приведенными ранее на рис. 4.10, так как фактически в таблицу Fio внесены все данные ранее созданного представления Abonent_Executor.

Приведенные выше запросы INSERT в Oracle и MS SQL Server реализуются аналогично.

В подзапросах, используемых в многострочном запросе INSERT, можно использовать оператор UNION. Пусть имеется таблица Phone со столбцами Abonent_Fio, Old_Tel и New_Tel, определенными

ми на типе данных соответственно VARCHAR(20), VARCHAR(15) и VARCHAR(15). Известно, что номера телефонов, которые начинались с «25», теперь начинаются с «14», а «68» сменились на «57». В таблицу Phone необходимо поместить ФИО абонентов, у которых изменился номер телефона, а также старые и новые номера телефонов (рис. 5.1). Для этого можно применить запрос

```
INSERT INTO Phone (Abonent_Fio, Old_Tel, New_Tel)
SELECT Fio, Phone, '14' || TRIM (LEADING '25' FROM Phone)
FROM Abonent WHERE SUBSTRING(Phone FROM 1 FOR 2) = '25'
UNION
SELECT Fio, Phone, '57' || TRIM (LEADING '68' FROM Phone)
FROM Abonent WHERE SUBSTRING(Phone FROM 1 FOR 2) = '68';
```

Данный запрос в Oracle и MS SQL Server реализуется аналогично, с использованием функций, свойственных для конкретной СУБД. Например, в Oracle:

```
INSERT INTO Phone (Abonent_Fio, Old_Tel, New_Tel)
SELECT Fio, Phone, '14' || LTRIM (PHONE,'25')
FROM Abonent WHERE SUBSTR(PHONE,1,2) = '25'
UNION
SELECT Fio, Phone, '57' || LTRIM (PHONE,'68')
FROM Abonent WHERE SUBSTR(PHONE,1,2) = '68';
```

ABONENT_FIO	OLD_TEL	NEW_TEL
Денисова Е. К.	680305	570305
Лукашина Р. М.	254417	144417
Маркова В. П.	683301	573301
Стародубцев Е. В.	683014	573014
Шубина Т. П.	257842	147842

Рис. 5.1. Таблица Phone после вставки

Ограничения

1. В подзапросе SELECT не следует использовать секцию ORDER BY, поскольку нецелесообразно сортировать таблицу результатов запроса, так как она добавляется в таблицу, которая, как и все остальные, не упорядочена.

2. В качестве значений, возвращаемых подзапросом SELECT, можно использовать агрегатные функции, но при этом нужно обеспечить совпадение типа и количества значений, возвращаемых запросом.

3. В секции FROM подзапроса SELECT нельзя использовать ту же самую таблицу, в которую запросом INSERT производится вставка строк.

Рассмотрим использование запроса INSERT с условием в Oracle. Пусть имеются три таблицы SmallNach, MediumNach, BigNach с набором столбцов, идентичным столбцам таблицы NachislSumma. Допустим, необходимо разделить все начисления из таблицы NachislSumma по величине начисленных сумм в три эти таблицы, а также увеличить год начисления на единицу для данных, вставляемых в таблицу BigNach. Запрос с условием можно записать так:

```
INSERT ALL
  WHEN NachislSum < 100 THEN
    INTO SMALLNACH
    VALUES (NachislFactCD, AccountCD, ServiceCD,
             NachislSum, NachislMonth, NachislYear)
  WHEN NachislSum >= 100 AND NachislSum < 500 THEN
    INTO MEDIUMNACH
    VALUES (NachislFactCD, AccountCD, ServiceCD,
             NachislSum, NachislMonth, NachislYear)
  ELSE
    INTO BigNach
    VALUES (NachislFactCD, AccountCD, ServiceCD,
             NachislSum, NachislMonth, NachislYear + 1)
SELECT NachislFactCD, AccountCD, ServiceCD,
       NachislSum, NachislMonth, NachislYear
FROM NachislSumma;
```

Содержание таблицы BigNach после вставки представлено на рис. 5.2.

NACHISLFACTCD	ACCOUNTCD	SERVICECD	NACHISLSUM	NACHISLMONTH	NACHISLYEAR
66	015527	4	580,1	7	2018
67	015527	4	611,3	10	2017
71	115705	4	553,85	1	2018
75	136169	4	528,44	10	2017
79	443690	4	500,13	9	2018

Рис. 5.2. Результат выполнения запроса INSERT с условием в Oracle

Рассмотрим пример множественного задания списка значений в секции VALUES запроса INSERT в MS SQL Server. Допустим, необходимо добавить сразу несколько строк в таблицу Abonent. В MS SQL Server для этого не требуется создавать несколько запросов INSERT, можно добавить все строки одним запросом:

```
INSERT INTO Abonent
VALUES ('500010', 8, 1, 5, 'Иванов В. И.', '551189'),
       ('500011', 8, 1, 6, 'Воськина Л. Г.', NULL),
       ('500012', 8, 1, 7, 'Мартынов И. Р.', '550303'),
       ('500013', 8, 1, 8, 'Федькин А. А.', '551102');
```

5.2. Обновление данных

Обновление значений как отдельных строк, так и всей таблицы или представления БД, осуществляется запросом-действием UPDATE. В общем виде запрос UPDATE указывает, где и что необходимо изменить, а также критерии отбора обновляемых строк:

```
UPDATE <таблица> SET {<столбец> = <выражение>}  
[WHERE <список условий>];
```

или

```
UPDATE <таблица> SET {<столбец> =  
  (SELECT <значение> FROM <откуда> WHERE <список условий>)}  
[WHERE <список условий_1>];
```

Обновление с помощью запроса UPDATE может быть позиционированным (выполняется только над одной строкой) и поисковым (выполняется над нулевым или большим количеством строк). Позиционированное изменение может появиться только в контексте текущей операции с курсором в модуле на PSQL, в то время как поисковое изменение появляется во всех остальных случаях. Поисковое изменение может эмулировать позиционированное изменение, если в секции WHERE задано условие, уникально определяющее строку (например, использующее первичный ключ таблицы). Запрос UPDATE может анализировать информацию из других таблиц БД, используя запрос SELECT (вложенный запрос).

В Firebird обобщенный формат запроса UPDATE следующий:

```
UPDATE { базовая_таблица | представление } [[AS] псевдоним]  
SET { столбец1 = {<значение1> | NULL | (<подзапрос1>)}  
  [, столбец2 = {<значение2> | NULL | (<подзапрос2>)} ] ... }  
[WHERE { <условие_поиска> | CURRENT OF имя_курсора } ]  
[PLAN <список_пунктов_плана>]  
[ORDER BY <элемент_сортировки1> [, <элемент_сортировки 2>]...]  
[ROWS k [TO r]]  
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

где

```
<значение> ::= { столбец | константа | <выражение> | функция }
```

Запрос UPDATE содержит секцию SET, в ней указывается изменение, которое нужно сделать для определенного столбца. Секция SET может включать столбцы лишь из обновляемой таблицы, т. е. значение одного или нескольких столбцов модифицируемой таблицы может быть заменено другим значением. В качестве значений могут применяться контекстные переменные (например, в контексте

текущего соединения клиента может использоваться контекстная переменная CURRENT_USER). Также можно установить для столбца NULL-маркер. Допускается применение вложенного запроса SELECT в секции SET.

Секция WHERE используется для отбора изменяемых строк. Условие поиска может использовать вложенный запрос SELECT. При отсутствии секции WHERE обновляются значения указанных столбцов во всех строках модифицируемой таблицы.

Секцию ORDER BY, задающую сортировку обрабатываемого набора данных, в запросе UPDATE целесообразно применять только в сочетании с секцией ROWS, которая ограничивает количество обновляемых строк указанным числом.

Запрос UPDATE, обновляющий только одну строку, может дополнительно включать секцию RETURNING для возврата значений обновляемой строки.

В Oracle запрос UPDATE представлен в следующем формате:

```
UPDATE { <целевой_объект> | ONLY (<целевой_объект>) } [псевдоним]
SET { столбец1 = {<значение1> | DEFAULT | (<подзапрос1>)}
    [, столбец2 = {<значение2> | DEFAULT | (<подзапрос2>)} ] ...
    | (столбец1 [,столбец2 ...]) = (<подзапрос>)
    | VALUE (псевдоним) = {<выражение> | (<подзапрос>)} }
[WHERE <условие_поиска>]
[RETURN[ING] <список_столбцов> [INTO <список_переменных>]];
```

где

```
<целевой_объект> ::=
{ таблица [[SUB]PARTITION { (имя_раздела)
  | FOR (ключ_раздела [, ключ_раздела ...]) }]
  | представление
  | (<запрос_SELECT> [WITH {READ ONLY | CHECK OPTION}
                    [CONSTRAINT имя_ограничения]])
  | TABLE (коллекция) [(+)] }
```

Формат указания целевого объекта в запросе UPDATE немного отличается от аналогичного синтаксиса в запросе INSERT (см. п. 5.1), если выполняется обновление секционированной таблицы. В конструкции [SUB]PARTITION можно задавать не имя раздела таблицы, а набор ключевых столбцов после слова FOR.

Ключевое слово ONLY используется только при обновлении представлений в случае, если целевое представление имеет зависимые представления, а требуется обновить только указанное представление.

В секции SET можно установить для столбца значение по умолчанию DEFAULT. Подзапрос в секции SET может представлять собой <скалярный_подзапрос>, если указан для обновления конкретного столбца, либо <подзапрос_столбца> (<подзапрос_строки>), если

указан для списка столбцов. Конструкция VALUE (псевдоним) в секции SET используется для обновления объектной таблицы Oracle.

В MS SQL Server запрос UPDATE имеет такой синтаксис [4]:

```
[ <предложение обобщенного табличного выражения> ]
UPDATE [ TOP (<выражение>) [ PERCENT ] ]
  { [<сервер>].[<база данных>.<схема>.]
    { <таблица> | <представление> }
    | { <функция OPENQUERY> | <функция OPENROWSET> } }
[ WITH (<подсказка таблицы> [, <подсказка таблицы>...] ) ]
SET <изменяемое значение> [, <изменяемое значение>]...
[<секция_OUTPUT> ]
[FROM (<таблица-источник> [, <таблица-источник>]...) ]
[WHERE { <условие_поиска> | CURRENT OF имя_курсора } ]
[OPTION (<подсказка запроса> [, <подсказка запроса>]...) ];
```

где

```
<изменяемое значение> ::=
  { <столбец> = { DEFAULT | NULL | <выражение> | (подзапрос) }
    | <столбец>.WRITE(<выражение>, @<смещение>, @<размер>)
    | @<переменная> [ = <столбец> ] <операция> <выражение> },
<операция> ::= += | -= | *= | /= | %= | &= | ^= | |= | =
```

В MS SQL Server секция SET, в которой задаются изменения, имеет особый формат. Существует несколько вариантов указания столбцов и новых их значений:

- в обычном варианте после имени столбца и знака равенства записывается константа, выражение, ключевое слово DEFAULT или NULL, скалярный подзапрос SELECT;
- вариант, когда после имени столбца задается обращение к функции .WRITE, может быть использован для столбцов с типом данных VARCHAR(MAX), NVARCHAR(MAX) и VARBINARY(MAX). Функция позволяет заменить данные в столбце, начиная с позиции, заданной локальной переменной @<смещение>, размером, заданным локальной переменной @<размер>, на значение выражения, указанного первым параметром функции;
- в третьем варианте используются локальные переменные. Им присваиваются конкретные значения. Эти же значения могут быть присвоены и столбцам изменяемой строки таблицы, когда конструкция представлена в виде @<переменная> [=<столбец>] <операция> <выражение>. Вначале столбцу таблицы присваивается значение, полученное в результате применения указанной операции к выражению, затем это же значение присваивается локальной переменной. Если в конструкции отсутствует имя столбца, то значение присваивается только локальной переменной.

Операция может иметь следующие значения:

- = (обычное присваивание значения);

- += (выполняется сложение указанного значения, полученного в результате вычисления выражения с существующим значением столбца, и результат присваивается переменной (и столбцу));
- -= (из значения вычитается полученное значение, и выполняется присваивание);
- *= (выполняется умножение и присваивание);
- /= (выполняется деление и присваивание);
- %= (получение остатка от деления и присваивание);
- &= (побитовая операция «И» и присваивание);
- ^= (побитовая операция исключающее «ИЛИ» и присваивание);
- |= (побитовая операция «ИЛИ» и присваивание).

Конструкция MS SQL Server @<переменная> [= <столбец>] <операция> <выражение> заменяет аналогичную конструкцию RETURNING <список_столбцов> [INTO <список_переменных>] в Firebird и Oracle.

Секция OUTPUT возвращает измененные строки таблицы.

Секция FROM предоставляет возможность создать для указания обновляемых строк высокоизбирательный критерий на основе соединения. Секция FROM позволяет использовать секцию JOIN, что значительно упрощает обновление строк целевой таблицы путем связывания строк из таблиц, указанных в секции FROM, со строками таблицы, обновляемой через UPDATE имя_таблицы. Секция FROM не нужна, если при указании строк используется только одна таблица — целевая.

Необязательная секция OPTION содержит подсказки оптимизатору запроса. Эти подсказки оказывают влияние на порядок выборки строк таблиц.

Примечание. Если при попытке обновить строку новое значение столбца противоречит ограничению, выдается сообщение об ошибке.

5.2.1. Явный запрос UPDATE

Явный запрос-действие UPDATE выбирает строки для изменения без использования подзапроса в секции WHERE. Запрос-действие UPDATE в Firebird имеет формат

```
UPDATE { базовая_таблица | представление}
SET столбец1 = {<значение1> | NULL} [, столбец2 = {<значение2> | NULL}
] ...
[WHERE <условие_поиска> | WHERE CURRENT OF имя_курсора]
[PLAN <список_пунктов_плана>]
[ORDER BY <элемент_сортировки1> [, <элемент_сортировки 2>]...]
[ROWS k [TO r]]
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

В данном случае в секции SET не используется подзапрос, секция WHERE применяется для отбора изменяемых строк без использования подзапроса, т. е. явно.

В Oracle явный запрос UPDATE имеет формат

```
UPDATE { <целевой_объект> | ONLY (<целевой_объект>) } [псевдоним]
SET { столбец1 = {<значение1> | DEFAULT } [, столбец2 = {<значение2> |
DEFAULT } ] ...
      | VALUE (псевдоним) = <выражение> }
[WHERE <условие_поиска>]
[RETURN[ING] <список_столбцов> [INTO <список_переменных>]];
```

В MS SQL Server явный запрос UPDATE имеет формат

```
UPDATE [ TOP (<выражение>) [ PERCENT ] ]
      { [<сервер>.] [<база данных>.]<схема>.]
      { <таблица> | <представление> }
      | { <функция OPENQUERY> | <функция OPENROWSET> } }
[WITH (<подсказка таблицы> [, <подсказка таблицы>...]) ]
SET { столбец1 = {DEFAULT | NULL | <выражение>}
      [, столбец2 = {DEFAULT | NULL | <выражение>} ] ...
      | столбец1.WRITE(<выражение>, @<смещение>, @<размер>)
      [, столбец2.WRITE(<выражение>, @<смещение>, @<размер>) ] ...
      | @<переменная> [ = <столбец1> ] <операция> <выражение>
      [, @<переменная> [ = <столбец2> ] <операция> <выражение> ] ... }
[<секция_OUTPUT> ]
[FROM (<таблица-источник> [, <таблица-источник>]...) ]
[WHERE { <условие_поиска> | CURRENT OF имя_курсора } ]
[OPTION (<подсказка запроса> [, <подсказка запроса>]...) ];
```

Рассмотрим примеры работы явного запроса-действия UPDATE.

Например, чтобы установить для всех абонентов номер телефона '982223', необходимо выполнить запрос

```
UPDATE Abonent SET Phone = '982223';
```

Чтобы изменить не все строки, а лишь те, которые удовлетворяют определенным условиям, необходимо задать это условие в секции WHERE запроса UPDATE. Предположим, что абонент с ФИО Шубина Т. П. (номер лицевого счета равен '080047') вышла замуж и у нее изменилась фамилия на Серова Т. П. Для внесения в таблицу Abonent соответствующих изменений нужно выполнить запрос

```
UPDATE Abonent SET Fio = 'Серова Т. П.'
WHERE AccountCD = '080047';
```

Таблица Abonent после обновления представлена на рис. 5.3.

Операция CASE может применяться не только в запросе SELECT. Она также очень полезна при изменении значений столбцов на ос-

нове условий, задаваемых полями строки, относящимися к другим столбцам. Примером такого использования операции CASE может быть запрос

```
UPDATE NachislSumma
SET NachislSum = CASE
    WHEN NachislYear = 2017
    THEN NachislSum*1.1
    WHEN NachislYear = 2018
    THEN NachislSum*1.2
END
WHERE ServiceCD = 1;
```

изменяющий значения начислений за услугу с кодом 1 за 2017 и 2018 гг. на 10 и 20 % соответственно.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
005488	3	4	1	Аксенов С. А.	556893
115705	3	1	82	Мищенко Е. В.	769975
015527	3	1	65	Конюхов В. С.	761699
443690	7	5	1	Тулупова М. И.	214833
136159	7	39	1	Свирина З. А.	<null>
443069	4	51	55	Стародубцев Е. В.	683014
136160	4	9	15	Шмаков С. В.	<null>
126112	4	7	11	Маркова В. П.	683301
136169	4	7	13	Денисова Е. К.	680305
080613	8	35	11	Лукашина Р. М.	254417
080047	8	39	36	Серова Т. П.	257842
080270	6	35	6	Тимошкина Н. Г.	321002

Рис. 5.3. Таблица Abonent после обновления

Приведенные выше запросы UPDATE корректно работают в любой из используемых в учебнике СУБД.

Рассмотрим задачу обновления значений типа BLOB в Firebird (см. параграф 2.8). Пусть в добавленный столбец Resume (подтип TEXT) таблицы Executor требуется внести, например резюме исполнителя ремонтных заявок. Для ввода такой информации исполнителю с кодом 1 нужно выполнить следующий запрос:

```
UPDATE Executor
SET Resume = 'Образование - среднее специальное; стаж - 15 лет.'
WHERE ExecutorCD = 1
```

Аргумент DEFAULT в MS SQL Server и Oracle можно также использовать для изменения значения столбца на NULL, если для него не определено значение по умолчанию и разрешены признаки NULL. Например, с помощью запроса

```
UPDATE Abonent SET Phone = DEFAULT;
```

номера телефонов у всех абонентов становятся неопределенными.

Рассмотрим ограничение количества обновляемых строк в запросе UPDATE. Допустим, необходимо установить номер телефона '982223' не для всех абонентов, а только для первых пяти абонентов с наименьшим номером лицевого счета. В Firebird запрос можно записать кратко:

```
UPDATE Abonent SET Phone = '982223'
ORDER BY AccountCD
ROWS 1 TO 5;
```

Microsoft SQL Server позволяет ограничивать количество строк, однако результат выполнения запроса вида

```
UPDATE TOP 5 Abonent SET Phone = '982223';
```

не будет совпадать с результатом работы запроса в Firebird, так как MS SQL Server не позволяет задавать сортировку. Аналогичный запрос в MS SQL Server может быть реализован только с использованием подзапроса в секции WHERE (подп. 5.2.2). В Oracle также не удастся реализовать аналогичный запрос без использования подзапроса.

Позиционированные обновления, использующие секцию WHERE CURRENT OF имя_курсора, будут подробно рассмотрены при изучении процедурного SQL (см. гл. 6).

5.2.2. Запрос UPDATE с подзапросом

В секции WHERE запроса UPDATE можно использовать вложенные запросы. Также допускается использование вложенного запроса в секции SET. Запрос-действие UPDATE с подзапросом в Firebird имеет формат

```
UPDATE {базовая_таблица | представление}
SET столбец1 = (<подзапрос1>) [, столбец2 = (<подзапрос2>) ] ...
    столбец1 [,столбец2 ...] = (<подзапрос>)
[WHERE { <условие_поиска_с_подзапросом> | CURRENT OF имя_курсора }]
[PLAN <список_пунктов_плана>]
[ORDER BY <элемент_сортировки1> [, <элемент_сортировки 2>]...]
[ROWS k [TO r]]
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

В секции SET <подзапрос> указывается для обновления определенного столбца и представляет собой <скалярный_подзапрос>. В <условие_поиска_с_подзапросом> секции WHERE подзапрос может представлять собой <скалярный_подзапрос>, <подзапрос_столбца> или <табличный_подзапрос> в зависимости от вида условия поиска (синтаксис условия поиска совпадает с описанным ранее при изучении вложенных запросов).

В Oracle подзапрос в запросе UPDATE, помимо секций SET и WHERE, может быть использован для указания целевого объекта для обновления. Формат запроса UPDATE с подзапросом в Oracle:

```
UPDATE { <целевой_объект> | ONLY (<целевой_объект>) } [псевдоним]
SET { столбец1 = (<подзапрос1>) [, столбец2 = (<подзапрос2>) ] ...
    | (столбец1 [,столбец2 ...]) = (<подзапрос>)
    | VALUE (псевдоним) = (<подзапрос>) }
[WHERE <условие_поиска_с_подзапросом>]
[RETURN[ING] <список_столбцов> [INTO <список_переменных>]];
```

В секции SET <подзапрос> представляет собой <скалярный_подзапрос>, если указан для обновления конкретного столбца, и <подзапрос_столбца> (<подзапрос_строки>), если указан для списка из нескольких столбцов.

В MS SQL Server запрос UPDATE имеет такой синтаксис [4]:

```
[ <предложение обобщенного табличного выражения> ]
UPDATE [ TOP (<выражение>) [ PERCENT ] ]
{ [<сервер>.] [<база данных>.]<схема>.]
{ <таблица> | <представление> }
| { <функция OPENQUERY> | <функция OPENROWSET> } }
[ WITH (<подсказка таблицы> [, <подсказка таблицы>...]) ]
SET столбец1 = (<подзапрос>) [, столбец2 = (<подзапрос>) ] ...
[<секция_OUTPUT> ]
[FROM (<таблица-источник> [, <таблица-источник>]...) ]
[WHERE { <условие_поиска_с_подзапросом> | CURRENT OF имя_курсора } ]
[OPTION (<подсказка запроса> [, <подсказка запроса>]...) ];
```

В секции SET <подзапрос> представляет собой <скалярный_подзапрос>, в секции WHERE — в зависимости от условия поиска.

Рассмотрим использование подзапроса в секции WHERE запроса UPDATE, которое дает возможность отбирать строки для обновления на основе информации из других таблиц. Например, для уменьшения на 10 значений начислений за услугу с кодом 2 абонентам, имеющим не более одной заявки на ремонт газового оборудования, нужно выполнить запрос

```
UPDATE NachislSumma N
SET NachislSum = NachislSum - 10
WHERE 2 > (SELECT COUNT (*)
          FROM Request R
          WHERE R.AccountCD = N.AccountCD)
AND ServiceCD = 2;
```

Фрагмент таблицы NachislSumma после обновления представлен на рис. 5.4.

Подзапрос, использованный в данном запросе UPDATE, является связанным, так как в нем используется столбец AccountCD внешней обновляемой таблицы (условие R.AccountCD = N.AccountCD). При выполнении обновления сначала выбирается строка из таблицы NachislSumma, затем выполняется вложенный запрос, возвращающий количество ремонтных заявок для выбранного абонента, а уже затем, в случае удовлетворения условию в секции WHERE запроса UPDATE, происходит обновление текущей строки таблицы NachislSumma. Обновление происходит последовательно для каждой строки таблицы NachislSumma без возврата к извлечению уже просмотренных строк.

В Oracle приведенный запрос может быть записан без изменений, в MS SQL Server запрос будет выглядеть так:

```
UPDATE NachislSumma
SET NachislSum = NachislSum - 10
WHERE 2 > (SELECT COUNT(*)
           FROM Request R
           WHERE R.AccountCD = NachislSumma.AccountCD)
AND ServiceCD = 2;
```

Приведем еще один пример с использованием соотнесенного вложенного запроса. Пусть требуется уменьшить на 10 значения начислений за сентябрь 2016 г. тем абонентам, которые до 15 сентября 2016 г. заплатили за август 2016 г. сумму, меньшую среднего значения всех своих платежей. Для решения этой задачи можно использовать запрос

```
UPDATE NachislSumma
SET NachislSum = NachislSum - 10
WHERE NachislYear = 2016
AND NachislMonth = 9
AND AccountCD IN (
  SELECT AccountCD
  FROM PaySumma A
  WHERE PayYear = 2016
  AND PayMonth = 8
  AND PayDate < '15.09.2016'
  AND PaySum < (
    SELECT AVG(PaySum)
    FROM PaySumma B
    GROUP BY AccountCD
    HAVING A.AccountCD = B.AccountCD
  )
);
```

Фрагмент таблицы NachislSumma после обновления представлен на рис. 5.5.

NACHISLFACTCD	ACCOUNTCD	SERVICECD	NACHISLSUM	NACHISLMONTH	NACHISLYEAR
...	...	...	...	...	...
10	080613	2	46,00	6	2016
...	...	...	...	...	...
20	015527	1	28,32	7	2017
...	...	...	...	...	...
22	080613	1	10,60	9	2017
...	...	...	...	...	...
24	015527	1	38,32	4	2018
...	...	...	...	...	...
26	080613	1	12,60	8	2015
...	...	...	...	...	...
29	136159	1	8,30	8	2018
...	...	...	...	...	...
32	443690	1	17,80	6	2017
...	...	...	...	...	...
34	126112	1	15,30	8	2015
...	...	...	...	...	...
36	080613	1	12,60	4	2017
...	...	...	...	...	...
40	015527	1	18,32	2	2017
41	443690	1	21,67	3	2018
42	080613	1	22,86	4	2015
...	...	...	...	...	...
46	126112	1	15,30	8	2016
...	...	...	...	...	...
48	136159	1	8,30	10	2017

Рис. 5.4. Таблица NachisSumma после уменьшения значений начислений

NACHISLFACTCD	ACCOUNTCD	SERVICECD	NACHISLSUM	NACHISLMONTH	NACHISLYEAR
...	...	...	...		...
5	115705	2	240,00	9	2016
...	...	...	...	...	...
17	443069	2	70,00	9	2016
...	...	...	...	...	...
47	443069	1	28,32	9	2016
...	...	...	...	...	...

Рис. 5.5. Содержание таблицы NachisSumma после обновления

В секции FROM подзапроса в запросе UPDATE может присутствовать имя обновляемой таблицы. В этом случае предполагается, что используется то состояние обновляемой таблицы, которое она имела до обновления.

Например, требуется уменьшить в 2 раза значения начислений за услугу с кодом 3, которые превышают среднее значение начислений за апрель 2017 г. по всем абонентам за эту же услугу. Для этого необходимо применить запрос UPDATE с подзапросом

```
UPDATE NachislSumma
SET NachislSum = NachislSum / 2
WHERE NachislSum > (
    SELECT AVG(NachislSum)
    FROM NachislSumma
    WHERE NachislMonth = 4
        AND NachislYear = 2017
        AND ServiceCD = 3
)
AND ServiceCD = 3;
```

Фрагмент таблицы NachislSumma после обновления представлен на рис. 5.6.

В предыдущем примере сначала выполнится подзапрос, возвратив среднее значение сумм начислений за апрель 2017 г., а затем произойдет обновление таблицы NachislSumma. В результате выполнения запроса в таблице NachislSumma 14 строк были обновлены.

Аналогично возможно использование соотнесенного вложенного запроса, относящегося к той же самой таблице, которая указана в качестве обновляемой. Рассмотрим пример такого соотнесенного подзапроса в секции SET. Например, для замены дат выполнения ремонтных заявок, относящихся к каждому абоненту, на наиболее позднюю дату их регистрации можно использовать запрос

```
UPDATE Request SET ExecutionDate =
(SELECT MAX (R.IncomingDate) FROM Request R
WHERE R.AccountCD = Request.AccountCD);
```

Первые 10 строк таблицы Request после обновления представлены на рис. 5.7.

После выполнения обновления у каждого абонента стала одинаковой дата выполнения всех ремонтных заявок (она стала совпадать с датой поступления последней заявки, поданной этим абонентом). Например, от абонента с номером лицевого счета '005488' поступило три ремонтных заявки: первая — 04.09.2015, вторая — 17.12.2016 и последняя — 04.04.2018. Дата выполнения всех трех заявок после выполнения запроса стала равной 04.04.2018.

NACHISLFACTCD	ACCOUNTCD	SERVICECD	NACHISLSUM	NACHISLMONTH	NACHISLYEAR
...	...	...	...	...	...
51	005488	3	139,90	5	2017
52	005488	3	133,35	2	2018
53	015527	3	171,68	11	2018
54	080047	3	135,80	2	2018
55	080270	3	139,12	11	2018
56	080613	3	127,20	7	2016
57	080613	3	129,40	2	2018
58	080613	3	119,66	5	2018
59	126112	3	179,90	4	2017
60	136159	3	90,06	9	2018
61	136160	3	119,40	3	2015
62	136160	3	118,69	3	2016
63	136169	3	174,59	6	2017
64	136169	3	173,09	7	2017
65	443690	3	145,16	3	2018
...	...	...	...	...	...

Рис. 5.6. Таблица NachisSumma после обновления

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
1	005488	1	1	17.12.2016	04.04.2018	True
2	115705	3	1	07.08.2016	28.12.2017	True
3	015527	1	12	28.02.2017	28.02.2017	False
5	080270	4	1	31.12.2016	31.12.2016	False
6	080613	1	6	16.06.2016	16.06.2016	True
7	080047	3	2	20.10.2017	11.10.2017	True
9	136169	2	1	06.11.2016	06.11.2016	True
10	136159	3	12	01.04.2016	01.04.2016	False
11	136160	1	6	12.01.2018	18.05.2018	True
12	443069	5	2	08.08.2016	13.09.2016	True

Рис. 5.7. Таблица Request после обновления

Вышеприведенные три запроса UPDATE корректны для всех анализируемых в учебнике СУБД. Рассмотрим некоторые особенности конкретных СУБД в части реализации запросов обновления с подзапросами.

В Oracle в секции SET можно использовать подзапрос не только для обновления значения конкретного столбца, но и для обновления значений сразу списка столбцов. Например, предыдущий запрос в Oracle можно расширить для замены не только дат выполнения ремонтных заявок, относящихся к каждому абоненту, на наиболее позднюю дату их регистрации, но также и дат поступления заявок на наиболее раннюю дату их регистрации:

```
UPDATE Request SET (IncomingDate, ExecutionDate) =  
  (SELECT MIN(R.IncomingDate), MAX(R.IncomingDate) FROM Request R  
   WHERE R.AccountCD = Request.AccountCD);
```

В Firebird и MS SQL Server такой запрос выполнить не удастся.

Главное расширение, которое вводит MS SQL Server в запрос UPDATE стандарта ANSI, — секция FROM. В следующем примере показано обновление данных в Firebird с помощью запроса UPDATE с довольно громоздким подзапросом, а потом — обновление с использованием секции FROM в запросе MS SQL Server.

Допустим, необходимо обновить данные в таблице Abonent, установив код улицы = 1 тем абонентам, на чьи ремонтные заявки был назначен исполнитель Школьников С. М. В Firebird запрос может быть записан так:

```
UPDATE Abonent  
SET StreetCD = 1 WHERE AccountCD IN  
  (SELECT AccountCD FROM Request WHERE ExecutorCD  
   IN (SELECT ExecutorCD FROM Executor  
       WHERE Fio = 'Школьников С. М.'));
```

В MS SQL Server запрос может быть записан проще:

```
UPDATE Abonent  
SET StreetCD = 1  
FROM Executor AS e  
JOIN Request AS r ON e.ExecutorCD = r.ExecutorCD  
WHERE r.AccountCD = Abonent.AccountCD AND e.Fio = 'Школьников С. М.';
```

Оба запроса выполняют одно и то же действие, но по-разному. Выполнение обновления в MS SQL Server сводится к соединению двух таблиц — Executor и Request — с таблицей Abonent. Для выполнения той же самой операции в запросе Firebird нужно сначала найти значение ExecutorCD в таблице Executor и передать его в таблицу Request, а затем найти значение AccountCD и передать его в основную инструкцию UPDATE. Таким образом, в MS SQL Server секция FROM значительно упрощает реализацию запроса UPDATE.

Однако не всегда секция FROM упрощает реализацию запроса обновления. Некоторые запросы могут быть проще реализованы в Firebird даже при использовании запроса UPDATE с секцией FROM в MS SQL Server. В подпараграфе 5.2.1 был рассмотрен пример простого запроса UPDATE в Firebird для установки номера телефона '982223' для первых пяти абонентов с наименьшим номером лицевого счета. В MS SQL Server аналогичный запрос может быть записан только с использованием подзапроса

```
UPDATE Abonent SET Phone = '982223'
FROM (SELECT TOP 5 * FROM Abonent ORDER BY AccountCD) AS A1
WHERE Abonent.AccountCD = A1.AccountCD;
```

Как правило, довольно сложно обновить первые N строк с помощью запроса UPDATE, если нет какой-нибудь явной последовательности строк, которую можно определить с помощью секции WHERE. В данном случае для получения первых пяти строк таблицы Abonent с наименьшим номером лицевого счета используется вложенный табличный подзапрос в секции FROM.

Следует отметить, что аналогичный запрос в Oracle также реализуется только с использованием подзапроса

```
UPDATE Abonent A SET A.Phone = '982223'
WHERE A.AccountCD IN
(SELECT A1.AccountCD
 FROM (SELECT A2.AccountCD FROM Abonent A2 ORDER BY A2.AccountCD) A1
 WHERE ROWNUM <= 5);
```

5.3. Удаление данных

Удаление всех, нескольких или одной строки из таблицы или представления осуществляется запросом-действием DELETE. Общий вид запроса DELETE простой: указывается, из какой таблицы удаляются данные, и, если требуется, отбираются удаляемые данные:

```
DELETE FROM <таблица> [WHERE <список условий>];
```

Существует два вида этого запроса: явный (простой) запрос DELETE и запрос DELETE с подзапросом. Удаление данных с помощью запроса DELETE может быть позиционированным и поисковым (аналогично описанному ранее обновлению данных с помощью запроса UPDATE).

В Firebird запрос имеет формат

```
DELETE FROM {базовая_таблица | представление} [ [ AS ] псевдоним]
[WHERE { <условие_поиска> | CURRENT OF имя_курсора } ]
[PLAN <список_пунктов_плана>]
[ORDER BY <элемент_сортировки1> [, <элемент_сортировки 2>]...]
```

```
[ROWS k [TO r]]
[RETURNING <список_столбцов> [INTO <список_переменных>]]];
```

С помощью DELETE можно удалять только отдельные строки, а не индивидуальные значения столбцов, поэтому параметр столбца является недоступным.

Платформа Oracle позволяет удалять строки из таблиц, представлений, материализованных представлений, вложенных подзапросов и секционированных представлений и таблиц запросом DELETE в формате [2, 3]

```
DELETE
[FROM] { <целевой_объект> | ONLY (<целевой_объект>) } [псевдоним]
[WHERE <условие_поиска> ]
[RETURNING <список_столбцов> [INTO <список_переменных>]]];
```

где

```
<целевой_объект> ::=
{ таблица [[SUB]PARTITION { (имя_раздела) | FOR (ключ_раздела
    [, ключ_раздела ...]) }}
| представление
| (<запрос_SELECT> [WITH {READ ONLY | CHECK OPTION}
    [CONSTRAINT имя_ограничения]])
| TABLE (коллекция) [(+)] }
```

Microsoft SQL Server поддерживает запрос DELETE в следующем виде:

```
[ <предложение обобщенного табличного выражения> ]
DELETE [ TOP (<выражение>) [ PERCENT ] ]
[FROM]
{ [<сервер>.] [[<база данных>.]<схема>.]
  { <таблица> | <представление> }
  | { <функция OPENQUERY> | <функция OPENROWSET> } }
[ WITH (<подсказка таблицы> [, <подсказка таблицы>...]) ]
[<секция_OUTPUT> ]
[FROM (<таблица-источник> [, <таблица-источник>]...) ]
[WHERE { <условие_поиска> | CURRENT OF имя_курсора } ]
[OPTION (<подсказка запроса> [, <подсказка запроса>]...) ];
```

Запрос DELETE в MS SQL Server может содержать подзапрос не только в секции WHERE, но также подзапрос для определения обобщенного табличного выражения.

Помимо запроса DELETE, Oracle и MS SQL Server поддерживают для полного удаления всех данных таблицы запрос TRUNCATE TABLE, который будет рассмотрен в подпараграфе 5.3.3.

Примечание. При попытке удалить строку, на значение которой имеется ограничение ссылочной целостности, выдается сообщение об ошибке.

5.3.1. Явный запрос DELETE

Явный запрос-действие DELETE позволяет удалить содержимое всех строк указанной таблицы или тех ее строк, которые выделяются условием поиска секции WHERE. Условие поиска явного запроса DELETE не содержит подзапроса. Например, для удаления всего содержимого таблицы Abonent можно выполнить запрос

```
DELETE FROM Abonent;
```

Обычно требуется удалить только определенные строки из таблицы или представления. Чтобы указать, какие строки нужно удалить, необходимо использовать условие поиска, как это делается для запроса SELECT. Так, чтобы удалить из таблицы Abonent абонента с номером лицевого счета, равным '005488', требуется выполнить запрос

```
DELETE FROM Abonent WHERE AccountCD = '005488';
```

Но в данном случае соответствующая строка из таблицы Abonent удалена не будет, так как имеются строки для данного абонента в дочерних таблицах (NachislSumma, PaySumma, Request).

Можно использовать DELETE с условием поиска, которое затрагивает несколько строк:

```
DELETE FROM NachislSumma WHERE NachislYear = 2015;
```

Вышеприведенные запросы DELETE корректно работают во всех трех рассматриваемых в учебнике СУБД.

Позиционированные удаления данных в Firebird, использующие секцию WHERE CURRENT OF имя_курсора, будут подробно рассмотрены при изучении процедурного языка SQL (гл. 6).

В общем случае правильно построить запрос на удаление довольно сложно: при малейшей неточности запрос DELETE удалит не те данные. Поэтому для предотвращения удаления лишних данных рекомендуется сначала убедиться, что удаляются только требуемые строки путем выборки их запросом SELECT. Это особенно важно при построении запросов DELETE с подзапросом.

5.3.2. Запрос DELETE с подзапросом

В секции WHERE запроса-действия DELETE можно использовать вложенные запросы, которые могут быть как независимыми, так и коррелированными.

Рассмотрим несколько примеров использования независимого подзапроса в секции WHERE запроса DELETE.

Для удаления ремонтных заявок, исполнителем которых является Булгаков Т. И., необходимо выполнить запрос

```
DELETE FROM Request WHERE ExecutorCD IN
  (SELECT ExecutorCD FROM Executor WHERE Fio = 'Булгаков Т. И.');
```

Первые восемь строк таблицы Request после удаления ремонтных заявок представлены на рис. 5.8.

Запрос DELETE удаляет все строки со значением столбца ExecutorCD исполнителя ремонтных заявок с ФИО Булгаков Т. И., возвращаемым подзапросом.

Особое место занимают запросы DELETE, где в основном запросе и подзапросе используется одна и та же таблица. В некоторых СУБД не рекомендуется использовать такие запросы.

С помощью запроса

```
DELETE FROM PaySumma

WHERE AccountCD IN (SELECT AccountCD
                    FROM PaySumma
                    GROUP BY AccountCD HAVING COUNT(PayFactCD) > 8);
```

можно удалить информацию о всех оплатах тех абонентов, которыми было произведено более восьми платежей.

Запрос вида

```
DELETE FROM Request WHERE IncomingDate NOT IN
  (SELECT MAX(IncomingDate) FROM Request GROUP BY AccountCD);
```

в таблице ремонтных заявок для каждого абонента оставляет только одну, самую позднюю заявку. Таблица Request после выполнения запроса представлена на рис. 5.9.

Подзапросы в секции WHERE могут иметь *несколько уровней вложенности*. Они могут также содержать внешние ссылки на целевую таблицу запроса DELETE. При этом единственное ограничение на применение подзапросов заключается в том, что *целевую таблицу нельзя указывать в секции FROM подзапроса независимо от уровня вложенности*. Это предотвращает ссылки из вложенных запросов на целевую таблицу (часть строк которой может быть удалена), за исключением внешних ссылок на строку, проверяемую в данный момент на соответствие условию поиска запроса DELETE.

Рассмотрим использование коррелированного подзапроса в секции WHERE запроса DELETE, т. е. подзапроса, который содержит внешнюю ссылку на текущую строку таблицы, из которой удаляются данные. Например, для удаления всех сведений об оплате услуг абонентами, проживающими на Татарской улице, можно использовать запрос

```
DELETE FROM PaySumma P WHERE EXISTS
  (SELECT * FROM Abonent A, Street S
   WHERE S.StreetNM = 'ТАТАРСКАЯ УЛИЦА' AND A.StreetCD = S. StreetCD
        AND P.AccountCD = A. AccountCD);
```


REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
1	005488	1	1	17.12.2016	20.12.2016	True
2	115705	3	1	07.08.2016	12.08.2016	True
3	015527	1	12	28.02.2017	08.03.2017	False
5	080270	4	1	31.12.2016	<null>	False
6	080613	1	6	16.06.2016	24.06.2016	True
7	080047	3	2	20.10.2017	24.10.2017	True
10	136159	3	12	01.04.2016	03.04.2016	False
11	136160	1	6	12.01.2018	12.01.2018	True

Рис. 5.8. Таблица Request после удаления ремонтных заявок

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
3	015527	1	12	28.02.2017	08.03.2017	False
5	080270	4	1	31.12.2016	<null>	False
6	080613	1	6	16.06.2016	24.06.2016	True
7	080047	3	2	20.10.2017	24.10.2017	True
9	136169	2	1	06.11.2016	08.11.2016	True
10	136159	3	12	01.04.2016	03.04.2016	False
11	136160	1	6	12.01.2018	12.01.2018	True
14	005488	4	6	04.04.2018	13.04.2018	True
18	115705	2	3	28.12.2017	04.01.2018	True
21	443069	1	2	13.09.2016	14.09.2016	True

Рис. 5.9. Таблица Request

В этом примере при проверке условия $P.AccountCD = A.AccountCD$ внутренний запрос ссылается на строку таблицы PaySumma, проверяемую в данный момент. Это означает, что подзапрос будет выполняться отдельно для каждой строки таблицы PaySumma. В результате выполнения запроса из таблицы PaySumma будет удалено 24 оплаты абонентами с номерами лицевого счетов '126112', '136160', '136169' и '443069' (именно они проживают на Татарской улице).

При реализации запросов DELETE с коррелированными подзапросами в MS SQL Server следует учитывать, что MS SQL Server не допускает определения псевдонима для таблицы, из которой удаляются данные. Таким образом, предыдущий запрос (как и все последующие) должен быть записан без псевдонима:

```
DELETE FROM PaySumma
WHERE EXISTS (
    SELECT *
    FROM Abonent A, Street S
    WHERE S.StreetNM = 'ТАТАРСКАЯ УЛИЦА'
    AND A.StreetCD = S.StreetCD
    AND PaySumma.AccountCD = A.AccountCD);
```

Для этого примера есть другой способ выполнить те же самые действия. Например, в Firebird:

```
DELETE FROM PaySumma P
WHERE 'ТАТАРСКАЯ УЛИЦА' IN
(SELECT StreetNM FROM Street S, Abonent A
WHERE A.StreetCD = S.StreetCD AND P.AccountCD = A.AccountCD);
```

В качестве подзапроса в секции WHERE запроса DELETE можно использовать запрос SELECT, который внутри себя содержит *соотнесенный подзапрос*. Например, требуется удалить ремонтные заявки тех абонентов, которые оплачивали услуги более восьми раз. Запрос будет выглядеть так:

```
DELETE
FROM Request R
WHERE R.AccountCD IN (
    SELECT P1.AccountCD
    FROM PaySumma P1
    WHERE 8 < (
        SELECT COUNT(*)
        FROM PaySumma P2
        WHERE P2.AccountCD = P1.AccountCD
    )
);
```

Таблица Request после удаления представлена на рис. 5.10.

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
1	005488	1	1	17.12.2016	20.12.2016	True
3	015527	1	12	28.02.2017	08.03.2017	False
5	080270	4	1	31.12.2016	<null>	False
6	080613	1	6	16.06.2016	24.06.2016	True
7	080047	3	2	20.10.2017	24.10.2017	True
9	136169	2	1	06.11.2016	08.11.2016	True
10	136159	3	12	01.04.2016	03.04.2016	False
11	136160	1	6	12.01.2018	12.01.2018	True
12	443069	5	2	08.08.2016	10.08.2016	True
13	005488	5	8	04.09.2015	05.12.2015	True
14	005488	4	6	04.04.2018	13.04.2018	True
19	080270	4	8	17.12.2016	27.12.2016	True
20	080047	3	2	11.10.2016	11.10.2016	True
21	443069	1	2	13.09.2016	14.09.2016	True
22	136160	1	7	18.05.2016	25.05.2016	True
23	136169	5	7	07.05.2016	08.05.2016	True

Рис. 5.10. Таблица Request после удаления ремонтных заявок абонентов, оплативших услуги более восьми раз

Для работы вложенного соотнесенного запроса формируется текущая строка-кандидат (берется первая строка таблицы PaySumma). Подзапрос возвращает количество оплат абонента с номером лицевого счета, содержащимся в строке-кандидате (условие P2.AccountCD=P1.AccountCD). Запрос DELETE удаляет все строки со значением столбца AccountCD, входящим во множество значений, формируемых вложенным соотнесенным запросом. В итоге будут удалены все строки таблицы Request, в которых значение столбца AccountCD равно 115705 (количество оплат, произведенных этим абонентом, равно 9). Существует и более простое решение данной задачи:

```
DELETE FROM Request A WHERE
(SELECT COUNT(*) FROM PaySumma B
WHERE B.AccountCD = A.AccountCD) > 8;
```

5.3.3. Удаление всех строк таблицы

Удалить все строки, не затронув структуру таблицы, в Oracle и MS SQL Server можно двумя способами:

- 1) если не использовать секцию WHERE в запросе DELETE (см. подпараграф 5.3.1);
- 2) использовать запрос TRUNCATE TABLE.

Удаление строк из большой таблицы, содержащей тысячи строк, является «дорогой» операцией, поскольку СУБД поддерживает копии всех удаленных транзакций на случай, если потребуется отменить внесенные изменения. Если таблица большая, то этот процесс может оказаться длительным и потребовать много места, пока операция удаления не завершится с помощью COMMIT.

Иногда нужно удалить все строки из таблицы, причем точно известно, что восстанавливать их не потребуется. В таких случаях вместо запроса DELETE можно использовать запрос TRUNCATE TABLE. При выполнении такого запроса СУБД удаляет все данные из таблицы, в том числе не сохраняет информацию, необходимую для отмены этих удалений. Когда СУБД усекает (truncate) таблицу, строки удаляются безвозвратно, однако структура таблицы остается. Общий вид запроса-действия TRUNCATE TABLE в Oracle:

```
TRUNCATE TABLE <таблица>;
```

Формат запроса в MS SQL Server:

```
TRUNCATE TABLE [[<база данных>.]<схема>.]<таблица>;
```

Запрос TRUNCATE TABLE работает быстрее, чем запрос удаления DELETE, и не журналируется. Также он не активирует триггеры, срабатывающие при удалении.

При этом следует учесть, что если на таблицу наложены условия, задающие внешние ключи (без использования опции ON DELETE

CASCADE), то нельзя усечь таблицу. Удаление не пройдет, даже если подчиненная таблица не имеет вообще никаких строк. Вначале следует отключить условие, используя запрос ALTER TABLE.

Например, можно удалить все строки таблицы Request так:

```
TRUNCATE TABLE Request;
```

Однако, например, таблицу Abonent усечь не получится из-за нарушений ссылочной целостности.

5.4. Условное манипулирование данными

5.4.1. Запрос UPDATE OR INSERT

Помимо запросов UPDATE и INSERT существует обобщенный запрос UPDATE OR INSERT, который предоставляет возможность изменять или вставлять строку в зависимости от того, существует она в целевой таблице или нет. Запрос UPDATE OR INSERT доступен для использования только в СУБД Firebird. Такой запрос имеет синтаксис

```
UPDATE OR INSERT
INTO { базовая_таблица | представление } [(<список_столбцов>)]
VALUES ({константа1 | переменная1} [, {константа2 | переменная2}] ...)
[MATCHING <список_столбцов>]
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

Например, требуется добавить исполнителя ремонтных заявок с ФИО Петров А. С. с кодом 1, причем если исполнитель с таким кодом уже существует, то следует изменить его ФИО на Петров А. С. (рис. 5.11). Запрос будет выглядеть так:

```
UPDATE OR INSERT INTO Executor VALUES (1,'Петров А. С.');
```

EXECUTORCD	FIO
1	Петров А. С.
3	Шубин В. Г.
4	Шлюков М. К.
5	Школьников С. М.

Рис. 5.11. Результат работы запроса UPDATE OR INSERT

Если бы исполнителя с кодом 1 не существовало в таблице Executor, в нее была бы вставлена новая строка.

Секция MATCHING используется для явного указания столбцов в таблице, значения в которых следует сопоставить с соответствующими значениями в секции VALUES. По умолчанию сопоставляются значения в столбцах первичных ключей.

Примечание. Если секция MATCHING не используется, то в целевой таблице должен обязательно существовать первичный ключ.

Например, предыдущий запрос с секцией MATCHING может быть записан как

```
UPDATE OR INSERT INTO Executor  
VALUES (1, 'Петров А. С.') MATCHING (ExecutorCD);
```

что подразумевается по умолчанию.

Рассмотрим пример, когда может быть полезным использование секции MATCHING. Например, требуется внести информацию о ремонтной заявке с кодом 25, поданной 17 декабря 2016 г. абонентом с номером лицевого счета '005488' и выполненной 22 декабря 2016 г. Запрос может выглядеть таким образом:

```
UPDATE OR INSERT INTO Request  
(RequestCD, AccountCD, IncomingDate, ExecutionDate)  
VALUES (25, '005488', '17.12.2016', '22.12.2016');
```

В процессе выполнения запроса происходит сравнение значения 25 со значениями столбца RequestCD (первичный ключ) в таблице Request. Совпадения не обнаруживается, поэтому происходит вставка новой строки в таблицу Request. Фрагмент таблицы Request после выполнения запроса представлен на рис. 5.12.

Предположим теперь, что вносятся данные по погашенной ремонтной заявке с кодом 26, поданной 28 февраля 2017 г. абонентом с номером лицевого счета '015527' и выполненной 10 марта 2017 г. При этом следует учесть, что если информация о заявке абонента с лицевым счетом '015527' от 28 февраля 2017 г. уже зарегистрирована, то необходимо обновить соответствующую строку. Для этого следует использовать запрос

```
UPDATE OR INSERT INTO Request  
(RequestCD, AccountCD, IncomingDate, ExecutionDate, Executed)  
VALUES (26, '015527', '28.02.2017', '10.03.2017', True)  
MATCHING (AccountCD, IncomingDate);
```

Фрагмент таблицы Request после выполнения запроса представлен на рис. 5.13.

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
...	...	...	...	...	...	...
25	005488	<null>	<null>	17.12.2016	22.12.2016	<null>

Рис. 5.12. Последняя строка в таблице Request после добавления

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
26	015527	1	12	28.02.2017	10.03.2017	True
...	...	...	...	...	...	...

Рис. 5.13. Обновленная строка в таблице Request

В процессе выполнения этого запроса значения первичных ключей не сравниваются, так как в секции MATCHING явно указано, что нужно сравнить значения в столбцах AccountCD и IncomingDate с соответствующими значениями из списка VALUES. Обнаруживается совпадение в строке таблицы Request с кодом заявки, равным 3, и происходит ее обновление (см. рис. 5.13).

В списке значений секции VALUES могут быть указаны не константы, а переменные, если запрос используется в модуле процедурного SQL.

Для выполнения данного запроса у пользователя должны быть права как на вставку, так и на изменение строк в таблице или представлении (управление доступом к ресурсам будет подробно описано далее).

5.4.2. Запрос MERGE

Для выбора строк из одной таблицы с целью обновления, удаления или вставки строк в другую таблицу используется запрос-действие MERGE [5, 6]. Выбор действия — обновить, удалить строку или добавить ее — определяется в зависимости от условия. Запрос MERGE фактически представляет собой объединение запросов INSERT, UPDATE и DELETE и позволяет избежать их многократного использования. Запрос MERGE используется во всех трех рассматриваемых в учебнике СУБД.

В Firebird запрос имеет синтаксис

```
MERGE
INTO <целевая_таблица> [ [AS] псевдоним ]
USING <исходная_таблица> [ [AS] псевдоним ] ON <условие_соединения>
[ WHEN MATCHED [ AND <условие> ] THEN
  { UPDATE SET столбец1 = <выражение1>
    [, столбец2 = <выражение2>] ... | DELETE} ]
[ WHEN NOT MATCHED [ AND <условие1> ] THEN
  INSERT [ ( <список_столбцов> ) ]
  VALUES ({константа1 | переменная1}
    [, {константа2 | переменная2}] ... ) ]
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

где

```
<целевая_таблица>:: = { базовая_таблица | представление},
<исходная_таблица> ::= { базовая_таблица | представление |
  <производная_таблица> | <хранимая_процедура>}
```

Запрос MERGE работает следующим образом. Выбирается первая строка из исходной таблицы и проверяется, существует ли такая строка целевой таблицы, что на ней становится истинным <условие_соединения>. Если да, то происходит переход к секции WHEN

MATCHED, проверяется дополнительное <условие> при его наличии и производится либо обновление этой строки целевой таблицы согласно запросу UPDATE, либо удаление этой строки (DELETE). Если таких строк целевой таблицы несколько, то все они обновляются или удаляются. Если для текущей строки исходной таблицы <условие_соединения> возвращает FALSE для всех строк целевой таблицы, то проверяется дополнительное <условие> при его наличии и в целевую таблицу вставляется строка согласно запросу INSERT, указанному в секции WHEN NOT MATCHED. В одном запросе MERGE могут быть определены секции WHEN MATCHED и WHEN NOT MATCHED одновременно или одно из них (обязательно). Допускается указывать несколько секций WHEN MATCHED и WHEN NOT MATCHED.

Описанная выше процедура повторяется для всех строк исходной таблицы.

Платформа Oracle поддерживает запрос MERGE с небольшими отклонениями от схемы синтаксиса ANSI в следующем формате:

```
MERGE
INTO <целевая_таблица> [ псевдоним ]
USING <исходная_таблица> [ псевдоним ] ON (<условие_соединения>)
[ WHEN MATCHED THEN
  { UPDATE SET столбец1 = { <выражение1> | DEFAULT }
    [, столбец2 = { <выражение2> | DEFAULT } ] ...
    [WHERE <условие>]
  | DELETE [WHERE <условие>] } [...] ]
[ WHEN NOT MATCHED THEN
  INSERT ( <список_столбцов> )
  VALUES ({ <выражение1> | DEFAULT } [,
    { <выражение2> | DEFAULT } ] ... ) ]
[WHERE <условие>];
```

К различиям между стандартом ANSI и его реализацией в Oracle относятся следующие:

- платформа Oracle не позволяет использовать ключевое слово AS для присвоения псевдонима целевой таблице или таблице-источнику;
- условие соединения должно быть обязательно заключено в скобки;
- в секции WHEN NOT MATCHED требуется указывать список вставляемых столбцов, хотя в стандарте ANSI это не является обязательным.

От синтаксиса Firebird запрос MERGE в Oracle отличается также местом указания дополнительных условий и возможностью задания в секции WHEN MATCHED двух действий — UPDATE и DELETE, выполняющихся в зависимости от этих условий.

В MS SQL Server синтаксис запроса MERGE следующий:

```

[ <предложение обобщенного табличного выражения> ]
MERGE [ TOP (<выражение>) [ PERCENT ] ]
[INTO] <целевая_таблица> [WITH ( <подсказка слияния> )]
[[AS] псевдоним ]
USING <исходная_таблица> ON <условие_соединения>
[ WHEN MATCHED [ AND <условие> ] THEN
  { UPDATE SET <изменяемое значение>
    [, <изменяемое значение>]... | DELETE } ] [...]
[ WHEN NOT MATCHED [ BY TARGET ] [ AND <условие> ] THEN
  INSERT [ ( <список_столбцов> ) ]
  { VALUES ( { DEFAULT | NULL | <выражение> }
    [, { DEFAULT | NULL | <выражение> } ]...)
    [, ( { DEFAULT | NULL | <выражение> }
    [, { DEFAULT | NULL | <выражение> } ]...)]...
    | DEFAULT VALUES}
[ WHEN NOT MATCHED BY SOURCE [ AND <условие> ] THEN
  { UPDATE SET <изменяемое значение>
    [, <изменяемое значение>]... | DELETE } ] [...]
[<секция_OUTPUT> ]
[OPTION (<подсказка запроса> [, <подсказка запроса>]...) ];
```

Запрос MERGE в MS SQL Server имеет ряд особенностей. Допускается использование двух секций WHEN MATCHED [7]. Если указаны две секции, то первая должна сопровождаться дополнительным условием (AND <условие>). Для любой строки вторая секция WHEN MATCHED применяется только в тех случаях, если не применяется первая. Если имеются две секции WHEN MATCHED, то одна должна указывать действие UPDATE, а другая — действие DELETE. Запрос MERGE нельзя применять для обновления одной строки более одного раза, а также использовать для обновления и удаления одной и той же строки.

Может быть указана секция WHEN NOT MATCHED BY SOURCE с действием UPDATE или DELETE. Запрос MERGE может иметь не более двух секций WHEN NOT MATCHED BY SOURCE. Если указаны две секции, то первая должна сопровождаться дополнительным условием. Для любой выбранной строки вторая секция WHEN NOT MATCHED BY SOURCE применяется только в тех случаях, если не используется первая. Если имеются две секции WHEN NOT MATCHED BY SOURCE, то одна должна указывать действие UPDATE, а другая — действие DELETE. В дополнительном условии можно ссылаться только на столбцы целевой таблицы.

Рассмотрим примеры использования запроса MERGE.

Допустим, требуется удалить все погашенные ремонтные заявки, поданные абонентами, проживающими на улице с кодом 7. Если у абонента нет зарегистрированных заявок, то следует внести информацию о заявке в таблицу Request. Запрос на такое слияние данных в Firebird:

```

MERGE
INTO Request R
USING (SELECT * FROM Abonent WHERE StreetCD = 7) Ab
  ON R.AccountCD = Ab.AccountCD
WHEN MATCHED AND R.Executed THEN
  DELETE
WHEN NOT MATCHED THEN
  INSERT (RequestCD, AccountCD, Executed)
  VALUES (24, Ab.AccountCD, False);

```

В Oracle можно составить аналогичный запрос, расширив его возможностью обновления всех непогашенных ремонтных заявок:

```

MERGE
INTO Request R
USING (SELECT * FROM Abonent WHERE StreetCD = 7) Ab
  ON (R.AccountCD = Ab.AccountCD)
WHEN MATCHED THEN
  UPDATE SET R.ExecutorCD = NULL WHERE R.Executed = 0
  DELETE WHERE R.Executed = 1
WHEN NOT MATCHED THEN
  INSERT (RequestCD, AccountCD, Executed)
  VALUES (24, Ab.AccountCD, '0');

```

В MS SQL Server первоначальный запрос будет корректен в таком виде:

```

MERGE
INTO Request R
USING (SELECT * FROM Abonent WHERE StreetCD = 7) Ab
  ON (R.AccountCD = Ab.AccountCD)
WHEN MATCHED AND R.Executed = 0 THEN
  DELETE
WHEN NOT MATCHED THEN
  INSERT (RequestCD, AccountCD, Executed)
  VALUES (24, Ab.AccountCD, 0);

```

Фрагмент таблицы Request после выполнения запроса представлен на рис. 5.14.

Производной таблицей, определенной в секции USING, возвращаются две строки, содержащие информацию об абонентах с номерами лицевых счетов '136159' и '443690', проживающих на улице с кодом 7. Поскольку у абонента со счетом '136159' зарегистрирована непогашенная ремонтная заявка с кодом 10, то строка из таблицы Request не удаляется. У абонента со счетом '443690' нет ни одной зарегистрированной заявки, поэтому строка, соответствующая данному абоненту, добавляется в таблицу Request.

Примером запроса с вычислением следующего по порядку значения первичного ключа может быть запрос, который для абонентов с начислениями более 500 увеличивает на 10 % значения соответствующих оплат, а при отсутствии оплат добавляет их с соответствующим значением начисления и датой оплаты, равной текущей:

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
...	...	...	...	...	...	...
10	136159	3	12	01.04.2016	03.04.2016	False
...	...	...	...	...	...	...
24	443690	<null>	<null>	<null>	<null>	False

Рис. 5.14. Обновленные и добавленные строки в таблице Request

```

MERGE INTO PaySumma P
USING (SELECT *
      FROM NachislSumma
      WHERE NachislSum > 500) AS N
ON (P.AccountCD = N.AccountCd
    AND P.ServiceCD = N.ServiceCD
    AND P.PayMonth = N.NachislMonth
    AND P.Payyear = N.NachislYear)
WHEN MATCHED THEN
  UPDATE SET PaySum = PaySum*1.1
WHEN NOT MATCHED THEN
  INSERT VALUES ((SELECT MAX(PayfactCD)+1 FROM PaySumma),
                 N.AccountCD, N.ServiceCD, N.NachislSum,
                 (SELECT CURRENT_DATE FROM RDB$DATABASE),
                 N.NachislMonth, N.NachislYear);

```

В результате выполнения этого запроса в таблицу PaySumma будет добавлена одна новая строка и в четырех строках увеличатся значения оплат.

Реализация данного запроса в Oracle и MS SQL Server отличается только особенностями получения текущей даты, а также отсутствием ключевого слова AS перед псевдонимом таблицы в Oracle.

Необходимо иметь в виду, что в обоих предыдущих примерах в таблицу добавляется только одна строка. В общем случае, если возможно добавление нескольких строк, запросы не выполняются из-за попытки вставки одинаковых значений в столбец первичного ключа. Для предотвращения такой ситуации при получении очередного уникального значения первичного ключа следует использовать генератор последовательности (см. подпараграф 6.1.5).

Следует отметить, что если запрос MERGE используется, например, в триггере Firebird совместно с секцией RETURNING, то результаты WHEN MATCHED UPDATE и WHEN NOT MATCHED INSERT доступны в контекстной переменной NEW.столбец, а результаты WHEN MATCHED DELETE — в OLD.столбец.

Запрос MERGE является удобным и мощным средством для синхронизации таблицы БД с внешними источниками данных.

5.5. Обновление представлений

Одной из операций над представлениями является их непосредственное использование с запросами модификации DML: INSERT, UPDATE и DELETE. Такие представления называются *модифицируемыми* (обновляемыми). При обновлении представлений изменения передаются в базовую таблицу, на которой основано представление.

Представление можно обновлять, если определяющий его запрос соответствует следующим требованиям:

- должен отсутствовать предикат DISTINCT, т. е. повторяющиеся строки не должны исключаться из НД;

— в секции FROM должна быть задана только одна таблица, которую можно обновлять, т. е. у представления должна быть одна исходная таблица, а пользователь должен иметь соответствующие права доступа к ней. Если исходная таблица сама является представлением, то оно также должно удовлетворять этим условиям;

— каждое имя в списке возвращаемых столбцов должно быть ссылкой на простой столбец, т. е. в этом списке не должны содержаться выражения, вычисляемые столбцы или агрегатные функции;

— секция WHERE не должна содержать вложенный запрос, т. е. в нем могут присутствовать только простые условия поиска;

— в запросе не должно содержаться секций GROUP BY или HAVING;

— для INSERT запрос должен включать столбцы основной таблицы с ограничением NOT NULL, если только не указано другое значение по умолчанию.

Эти требования базируются на том принципе, что представление разрешается обновлять в том случае, если СУБД может для каждой строки представления найти исходную строку в исходной таблице, а для каждого обновляемого столбца представления — исходный столбец в исходной таблице. Если представление соответствует этим требованиям, то над ним и, как следствие, над исходной таблицей можно выполнять имеющие смысл операции вставки, удаления и обновления. Например, следующие представления являются представлениями только для чтения:

```
/* Представление DailyRequest только для чтения из-за наличия DISTINCT в запросе */
```

```
CREATE VIEW DailyRequest  
AS SELECT DISTINCT AccountCD, RequestCD, IncomingDate, FailureCD  
FROM Request;
```

```
/* Представление SummTotal только для чтения из-за наличия агрегатной функции в списке возвращаемых элементов, обращения к двум таблицам и использования секции GROUP BY */
```

```
CREATE VIEW SummTotal (Abonent, Summa)  
AS SELECT Fio, SUM(NachislSum) FROM Abonent A, NachislSumma N  
WHERE A.AccountCD = N.AccountCD  
GROUP BY Fio;
```

```
/* Представление Summ_Abonent только для чтения из-за наличия вложенного запроса в запросе SELECT */
```

```
CREATE VIEW Summ_Abonent  
AS SELECT * FROM Abonent WHERE AccountCD IN  
(SELECT AccountCD FROM NachislSumma WHERE NachislSum = 46);
```

Пояснения, стоящие перед запросами CREATE VIEW, представляют собой комментарии SQL и будут подробно рассмотрены далее при изучении процедурного языка (подпараграф 6.1.1).

Рассмотрим примеры работы с обновляемыми представлениями, которые наглядно демонстрируют, как изменяются данные представления при манипулировании данными таблицы, на основе которой оно создано, и как изменяются данные таблицы при манипулировании данными представления. Результаты выполнения будут приведены для СУБД Firebird, при этом все рассматриваемые запросы одинаково корректно работают и в Oracle, и в MS SQL Server, отличаясь в некоторых случаях только последовательностью вывода строк результата.

Пусть требуется создать смешанное представление, которое содержит снимок таблицы Abonent. Вертикальное подмножество должно включать в себя столбцы с ФИО и номерами лицевых счетов абонентов, а горизонтальное подмножество — строки со значением номера лицевого счета абонента, большего или равного 200 000. Для этого можно использовать запрос

```
CREATE VIEW Abonent_View (AccountCD, Fio)
AS SELECT AccountCD, Fio FROM Abonent
WHERE AccountCD >= '200000' WITH CHECK OPTION;
```

Результат выполнения запроса

```
SELECT * FROM Abonent_View;
```

представлен на рис 5.15. Это представление является обновляемым, если:

- 1) является подмножеством одной таблицы (нет соединения таблиц);
- 2) в списке возвращаемых элементов нет вычисляемых выражений и агрегатных функций;
- 3) используется простое условие поиска (без подзапросов);
- 4) запрос SELECT, на котором базируется представление, не содержит DISTINCT, секций GROUP BY и HAVING.

ACCOUNTCD	FIO
443690	Тулупова М. И.
443069	Стародубцев Е. В.

Рис. 5.15. Данные представления Abonent_View

Следовательно, к этому представлению могут быть применены запросы INSERT, UPDATE и DELETE. Продемонстрируем это с помощью следующих примеров. Пусть требуется вставить строку в набор данных, возвращаемый представлением Abonent_View. Это можно выполнить с помощью запроса


```
INSERT INTO Abonent_View (AccountCD, Fio)
VALUES ('999999', 'Васильев С. В.');
```

Результат выборки всех данных из представления Abonent_View представлен на рис. 5.16.

Данные, содержащиеся в таблице Abonent после применения операции вставки в представление Abonent_View, приведены на рис. 5.17.

ACCOUNTCD	FIO
443690	Тулупова М. И.
443069	Стародубцев Е. В.
999999	Васильев С. В.

Рис. 5.16. Данные представления Abonent_View после вставки

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
005488	3	4	1	Аксенов С. А.	556893
115705	3	1	82	Мищенко Е. В.	769975
015527	3	1	65	Конюхов В. С.	761699
443690	7	5	1	Тулупова М. И.	214833
136159	7	39	1	Свирина З. А.	<null>
443069	4	51	55	Стародубцев Е. В.	683014
136160	4	9	15	Шмаков С. В.	<null>
126112	4	7	11	Маркова В. П.	683301
136169	4	7	13	Денисова Е. К.	680305
080613	8	35	11	Лукашина Р. М.	254417
080047	8	39	36	Шубина Т. П.	257842
080270	6	35	6	Тимошкина Н. Г.	321002
999999	<null>	<null>	<null>	Васильев С. В.	<null>

Рис. 5.17. Данные таблицы Abonent

Следует отметить важное обстоятельство: с помощью созданного представления возможна вставка значений в столбцы AccountCD и Fio и предотвращается вставка определенных значений в остальные столбцы. Открывая доступ к этому представлению, разрешается пользователям и программам вставлять данные только в эти два столбца. Пользователи получают возможность создавать но-

вые строки в таблице Abonent, вставляя данные в представление Abonent_View, но они смогут размещать значения только в двух столбцах, которые заданы в определении представления.

Проверим работу опции WITH CHECK OPTION, которая должна ограничивать операции DML, применяемые к представлению таким образом, чтобы они не нарушали условия поиска запроса SELECT, на котором базируется представление. Начнем с INSERT. Для этого выполним запрос

```
INSERT INTO Abonent_View (AccountCD, Fio)
VALUES ('100000', 'Скляров А. А.');
```

Этот запрос на вставку не будет выполнен, так как содержит вставку значений, нарушающих условие отбора запроса SELECT, с помощью которого представление было создано. Будет выдано сообщение об ошибке, в Firebird такого содержания: «Operation violates CHECK constraint on view or table ABONENT_VIEW» (нарушение ограничения CHECK представления Abonent_View).

Если бы при определении представления Abonent_View не было опции WITH CHECK OPTION, то предыдущий запрос не вернул бы ошибку. Строка с номером лицевого счета '100000' была бы вставлена в таблицу Abonent, на которой базируется представление, но в самом представлении ее не было бы видно. Чтобы избежать таких противоречий, нужно включать в определение представления опцию WITH CHECK OPTION. Тогда до реального выполнения операций модификации или вставки строк через представление для каждой строки будет проверяться, соответствует ли она условиям представления.

Выполним запрос на удаление:

```
DELETE FROM Abonent_View WHERE AccountCD = '999999';
```

Данные, содержащиеся в таблице Abonent после удаления строки через представление Abonent_View, приведены на рис. 5.18.

Рассмотрим еще один пример работы с обновляемым представлением.

Пусть необходимо создать представление, которое показывало бы всю информацию об абонентах, кроме информации о номере телефона. При этом должны быть исключены из рассмотрения абоненты с фамилиями, начинающимися с буквы М. Запрос на создание такого представления выглядит так:

```
CREATE VIEW Abonent_Information (Code, Street, House, Flat, Name)
AS SELECT AccountCD, StreetCD, HouseNo, FlatNo, Fio
FROM ABONENT
WHERE Fio NOT LIKE 'М%' WITH CHECK OPTION;
```

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
005488	3	4	1	Аксенов С. А.	556893
115705	3	1	82	Мищенко Е. В.	769975
015527	3	1	65	Конюхов В. С.	761699
443690	7	5	1	Тулупова М. И.	214833
136159	7	39	1	Свирина З. А.	<null>
443069	4	51	55	Стародубцев Е. В.	683014
136160	4	9	15	Шмаков С. В.	<null>
126112	4	7	11	Маркова В. П.	683301
136169	4	7	13	Денисова Е. К.	680305
080613	8	35	11	Лукашина Р. М.	254417
080047	8	39	36	Шубина Т. П.	257842
080270	6	35	6	Тимошкина Н. Г.	321002

Рис. 5.18. Данные таблицы Abonent после удаления

Это представление является обновляемым, потому что запрос основан на одной таблице, каждый столбец представления ссылается на простой столбец и секция WHERE не содержит вложенного запроса. Данные, которые пользователь увидит после выполнения запроса

```
SELECT * FROM Abonent_Information;
```

представлены на рис. 5.19.

CODE	STREET	HOUSE	FLAT	NAME
005488	3	4	1	Аксенов С. А.
015527	3	1	65	Конюхов В. С.
443690	7	5	1	Тулупова М. И.
136159	7	39	1	Свирина З. А.
443069	4	51	55	Стародубцев Е. В.
136160	4	9	15	Шмаков С. В.
136169	4	7	13	Денисова Е. К.
080613	8	35	11	Лукашина Р. М.
080047	8	39	36	Шубина Т. П.
080270	6	35	6	Тимошкина Н. Г.

Рис. 5.19. Данные представления Abonent_Information

Допустим, что был выполнен запрос на изменение данных в таблице Abonent:

```
UPDATE Abonent SET Fio = 'Булгакова Т. П.' WHERE AccountCD = '080047';
```

Пользователем также был выполнен запрос на добавление данных в представление Abonent_Information:

```
INSERT INTO Abonent_Information (Code, Street, House, Flat, Name)
VALUES ('123456', 3, 12, 34, 'Тарасов А. В.');
```

Поскольку столбец Phone может содержать NULL, то вставка строки в представление проходит успешно (рис. 5.20):

```
SELECT * FROM Abonent_Information;
```

CODE	STREET	HOUSE	FLAT	NAME
005488	3	4	1	Аксенов С. А.
015527	3	1	65	Конюхов В. С.
443690	7	5	1	Тулупова М. И.
136159	7	39	1	Свирина З. А.
443069	4	51	55	Стародубцев Е. В.
136160	4	9	15	Шмаков С. В.
136169	4	7	13	Денисова Е. К.
080613	8	35	11	Лукашина Р. М.
080047	8	39	36	Булгакова Т. П.
080270	6	35	6	Тимошкина Н. Г.
123456	3	12	34	Тарасов А. В.

Рис. 5.20. Данные представления Abonent_Information после вставки строки и обновления таблицы Abonent

Пользователь может выполнять запросы на обновление и удаление данных в представлении. Допустим, были выполнены последовательно два запроса:

```
UPDATE Abonent_Information SET Name = 'Симкина З. А.'
WHERE Code = '136159';
DELETE FROM Abonent_Information WHERE Code = '123456';
```

Данные, получаемые в результате выполнения запроса

```
SELECT * FROM Abonent_Information;
```

представлены на рис. 5.21.

CODE	STREET	HOUSE	FLAT	NAME
005488	3	4	1	Аксенов С. А.
015527	3	1	65	Конюхов В. С.
443690	7	5	1	Тулупова М. И.
136159	7	39	1	Симкина З. А.
443069	4	51	55	Стародубцев Е. В.
136160	4	9	15	Шмаков С. В.
136169	4	7	13	Денисова Е. К.
080613	8	35	11	Лукашина Р. М.
080047	8	39	36	Булгакова Т. П.
080270	6	35	6	Тимошкина Н. Г.

Рис. 5.21. Данные представления Abonent_Information после обновления и удаления

Если пользователь попытается вставить или обновить значение столбца Name для абонента с номером лицевого счета '005488', установив ФИО абонента, например, как Морозов А. Н., с помощью запроса

```
UPDATE Abonent_Information SET Name = 'Морозов А. Н.'
WHERE Code = '005488';
```

то подобный запрос потерпит неудачу из-за того, что при создании представления была использована опция WITH CHECK OPTION. Будет выдано сообщение об ошибке: «Operation violates CHECK constraint on view or table Abonent_Information» (нарушение ограничения CHECK представления Abonent_Information).

Если пользователь выполнит запрос на удаление

```
DELETE FROM Abonent_Information WHERE Name LIKE 'М%';
```

то ошибки не возникнет, но в самой таблице Abonent никаких изменений не произойдет (в данном случае это не является следствием использования опции WITH CHECK OPTION, а есть следствие применения в запросе представления условия поиска WHERE Fio NOT LIKE 'М%').

Чтобы пояснить результаты действия опции WITH CHECK OPTION, рассмотрим еще один пример. Допустим, что в БД созданы два обновляемых представления: Nach_Information_20, показывающее информацию о начислениях со значениями, меньшими 20 (идентификатор факта начисления, номер лицевого счета абонента и значение начисления), и Nach_Information_15, отображающее данные предыдущего представления со значениями более 15. Запросы на создание этих двух представлений:

```
CREATE VIEW Nach_Information_20 (Fact, Code, Summa)
AS SELECT NachislFactCD, AccountCD, NachislSum
    FROM NachislSumma WHERE NachislSum<20 WITH CHECK OPTION;
CREATE VIEW Nach_Information_15
AS SELECT Fact, Code, Summa
    FROM Nach_Information_20 WHERE Summa >15 WITH CHECK OPTION;
```

Данные, получаемые после выполнения запроса

```
SELECT * FROM Nach_Information_20;
```

представлены на рис. 5.22.

ФАКТ	CODE	SUMMA
6	136160	18,30
21	080047	19,56
22	080613	10,60
26	080613	12,60
29	136159	8,30
32	443690	17,80
34	126112	15,30
36	080613	12,60
40	015527	18,32
48	136159	8,30
50	136160	18,30

Рис. 5.22. Данные представления Nach_Information_20

Данные, получаемые после выполнения запроса

```
SELECT * FROM Nach_Information_15;
```

представлены на рис. 5.23.

ФАКТ	CODE	SUMMA
6	136160	18,30
21	080047	19,56
32	443690	17,80
34	126112	15,30
40	015527	18,32
50	136160	18,30

Рис. 5.23. Данные представления Nach_Information_15

В каждом из представлений Nach_Information_20 и Nach_Information_15 может отсутствовать или присутствовать опция WITH CHECK OPTION. Рассмотрим, что будет происходить в каждом из возможных случаев при выполнении следующих двух запросов на модификацию строк (будем называть эти запросы соответственно U1 и U2):

```
UPDATE Nach_Information_15 SET Summa = Summa + 5;  
UPDATE Nach_Information_15 SET Summa = Summa - 5;
```

Случай 1. Ни в одном из представлений не содержится опция WITH CHECK OPTION.

Допустим, был выполнен запрос U1 на увеличение начисленных сумм. После выполнения запроса U1 представление Nach_Information_15 оказывается пустым. В таблице NachislSumma в тех строках, которые попадали в представление Nach_Information_15 (см. рис. 5.23), будут увеличены на 5 значения начислений (NachislSum). В результате значения в этих строках превысят 20, и строки, перестав удовлетворять условию представления Nach_Information_20, исчезнут из него.

Данные представления Nach_Information_20 после выполнения запроса U1 показаны на рис. 5.24.

ФАКТ	CODE	SUMMA
22	080613	10,60
26	080613	12,60
29	136159	8,30
36	080613	12,60
48	136159	8,30

Рис. 5.24. Данные представления Nach_Information_20 после выполнения запроса U1

Этот результат может быть неожиданным для пользователей БД, которым известно, что условие представления Nach_Information_15 имеет вид Summa > 15 и соблюдение этого условия должно сохраняться при увеличении размера начисления. В то же время в таблице NachislSumma обновляются значения столбца NachislSum для шести строк (с идентификаторами факта начисления 6, 21, 32, 34, 40, 50).

После выполнения запроса U2 представление Nach_Information_15 также оказывается пустым. Данные представления Nach_Information_20 после выполнения запроса U2 показаны на рис. 5.25.

Возможно, результат будет достаточно естественным для пользователей представления Nach_Information_15, которым известно

условие представления, но те, кто работает с представлением Nach_Information_20, обнаружат в теле результирующей таблицы шесть строк с измененными значениями в столбце Summa (с идентификаторами факта начисления 6, 21, 32, 34, 40, 50).

FACT	CODE	SUMMA
6	136160	13,30
21	080047	14,56
22	080613	10,60
26	080613	12,60
29	136159	8,30
32	443690	12,80
34	126112	10,30
36	080613	12,60
40	015527	13,32
48	136159	8,30
50	136160	13,30

Рис. 5.25. Данные представления Nach_Information_20 после выполнения запроса U2

В таблице NachislSumma также будут изменены шесть строк.

Случай 2. В определении представления Nach_Information_20 содержится опция WITH CHECK OPTION, а в определении Nach_Information_15 опция WITH CHECK OPTION отсутствует. В этом случае запрос U1 будет отвергнут системой (поскольку его выполнение нарушает условие представления Nach_Information_20). Будет выдано следующее сообщение: «Operation violates CHECK constraint on view or table NACH_INFORMATION_20» (нарушение ограничения CHECK представления).

Но заметим, что такое поведение системы будет совершенно неожиданным и непонятным для тех пользователей БД, которым известно только определение представления Nach_Information_15, поскольку запрос U1 явно не может нарушить видимое ими ограничение.

В то же время запрос U2 будет выполнен и по-прежнему приведет к опустошению представления Nach_Information_15. Те, кто работает с представлением Nach_Information_20, обнаружат в теле результирующей таблицы шесть строк с измененными значениями столбца NachSum, как и в предыдущем случае (см. рис. 5.25).

Случай 3. В определении представления Nach_Information_20 опция WITH CHECK OPTION отсутствует, а в определении Nach_Information_15 содержится опция WITH CHECK OPTION.

Запрос U1 (увеличение размера начисления) будет успешно выполнен, поскольку он не противоречит локальным ограничениям представления Nach_Information_15. После выполнения запроса U1 представление Nach_Information_15 оказывается пустым. Его строки перестают удовлетворять условию представления Nach_Information_20 и исчезают из него. Данные представления Nach_Information_20 будут совпадать с данными, показанными на рис. 5.24 (случай 1).

В таблице NachislSumma также будут изменены шесть строк.

Запрос U2 не будет работать (его выполнение не будет допущено условием проверки представления Nach_Information_15).

Случай 4. В определении представлений Nach_Information_20 и Nach_Information_15 содержится опция WITH CHECK OPTION.

В этом случае запрос U1 не будет выполнен, так как его выполнение нарушает условие проверки представления Nach_Information_20, а запрос U2 не будет выполнен, так как его выполнение нарушает условие проверки представления Nach_Information_15.

Только в этом случае операции обновления будут выполняться корректно. Очевидный вывод из приведенного анализа заключается в том, что единственным способом обеспечить корректность выполнения операций обновления через представления (допускающие операции обновления) является включение в определение каждого имеющегося в БД представления опции WITH CHECK OPTION.

Контрольные вопросы

1. Какие запросы языка DML используются в SQL СУБД Firebird?
2. Как осуществляется добавление новых данных в таблицу с помощью однострочного запроса?
3. Как добавить в таблицу строки из другой таблицы?
4. Какие существуют ограничения на подзапрос, используемый в многострочном запросе INSERT?
5. Как осуществляется простое обновление данных в таблице БД?
6. Как могут использоваться подзапросы при обновлении данных?
7. Как построить запрос на обобщенное добавление и обновление данных средствами языка SQL?
8. Как выполнить обновление или вставку строк в одной таблице на основании данных другой таблицы?
9. Как выполнить простое удаление данных из таблицы БД?
10. Каким образом работает запрос на удаление данных с подзапросом?
11. В каком случае к представлению можно применять запросы DML?
12. Какие проблемы могут возникнуть, если при создании представления не использовалась секция WITH CHECK OPTION?

Глава 6

ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ

Эта глава посвящена изучению процедурного языка SQL обработки данных, который предполагает расширение стандартного набора запросов, изученных в предыдущих главах, средой программирования. Практически в каждой современной СУБД используется свой процедурный язык, позволяющий обрабатывать возникающие проблемы, создавать и выполнять ХП, триггеры и сценарии с использованием локальных переменных, операторов присваивания, ветвления и циклов. Язык, который предоставляет такую возможность для сервера Firebird, называется Procedural SQL (PSQL), в СУБД Oracle Database это Procedural Language/SQL (PL/SQL), в MS SQL Server — Transact-SQL (T-SQL), в PostgreSQL — Procedural Language/PostgreSQL Structured Query Language (PL/pgSQL), в MySQL — SQL/Persistent Stored Module (SQL/PSM). Основным назначением процедурного языка являются создание серверной логики приложений и более гибкая обработка данных. Например, PL/SQL также является основным языком программирования таких продуктов, как Oracle Forms (создание экранных форм) и Oracle Reports (создание отчетов).

В предыдущих главах запросы SQL использовались (выполнялись) в интерактивном режиме (см. параграф 2.4). В комбинации с операторами процедурного SQL появляется возможность автоматизировать процесс обработки данных, в частности, с целью получения новой информации. Применительно к рассматриваемой в настоящем учебнике предметной области это может быть информация о дисциплине оплаты жилищно-коммунальных услуг, а также анализ выполнения ремонтных заявок.

Процедурный язык — это простой, но достаточно мощный набор процедурных расширений языка SQL, которые были включены еще в 1999 г. в стандарт языка SQL3. В каждой СУБД, использующей процедурный язык, существует возможность создания независимых исполняемых модулей, представленных разработчиками в виде исходных кодов: ХП, функций, триггеров, выполнимых блоков, процедурных пакетов. Такие коды выполняются полностью на сервере, возвращая при необходимости клиентскому приложению значения, полученные в результате выполнения.

Хранимые процедуры и триггеры представляют собой группы связанных между собой операторов процедурного SQL и запросов SQL, применение которых делает работу разработчика более легкой и гибкой, поскольку часто оказывается гораздо проще выполнить ХП, чем последовательность отдельных команд и запросов. Хранимые процедуры сохраняются в БД в откомпилированном виде (в виде исполняемых модулей). Фактически запросы, написанные на декларативном языке, вкладываются в исполняемые модули, написанные на императивном языке (рис. 6.1).

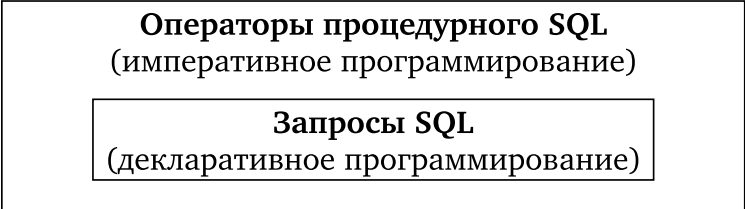


Рис. 6.1. Языковая структура исполняемого модуля

Хранение процедур в том же месте, где они исполняются, обеспечивает уменьшение объема передаваемых по сети данных и повышает общую производительность системы. Применение хранимых процедур упрощает сопровождение программных комплексов и внесение изменений в них, позволяет повысить независимость прикладного программного обеспечения. Например, изменение схемы контроля в триггере автоматически отражается во всех приложениях, не требуя внесения в них каких-либо изменений.

Применение активных баз данных в информационных системах позволяет перенести управляющую бизнес-логику в СУБД, которая будет сама обнаруживать возникновение заранее описанных событий в объектах контроля и управления и реагировать на них в соответствии с заранее определенными действиями.

Таким образом, СУБД может стать не просто центром хранения и обработки информации от объектов контроля, но также центром принятия решений и выработки управляющих воздействий на объекты управления.

Активные СУБД объединяют технологии БД с программированием «на логических правилах» для придания БД возможности реагирования на некоторые (возможно, внешние) события.

Активность — это способность БД выполнять некоторые действия над данными опосредованно, без явного вмешательства пользователя. Присущие активным БД возможности реагирования на события можно использовать в широком спектре приложений, включающих безопасность, материализацию представлений, проверку целостности и т. д.

Активная база данных состоит из (пассивной) БД и множества активных правил (см. параграф 1.8).

В настоящей главе, прежде всего, рассмотрен ряд предварительных вопросов, относящихся к процедурным языкам изучаемых СУБД, без знания которых невозможно создавать и использовать модули на PSQL, T-SQL или PL/SQL, приведены особенности создания и работы с различными модулями на процедурных языках различных СУБД.

6.1. Основы разработки исполняемых модулей

Процедурный язык — это блочно-структурированный язык программирования, используемый для написания различных исполняемых модулей в СУБД. Исполняемые модули могут быть именованными объектами БД — это ХП, функции, триггеры, пакеты — или неименованными блоками.

Процедурный язык может включать стандартные запросы SQL, а также расширения языка SQL. Несмотря на то, что стандартный язык SQL предоставляет огромный набор средств для взаимодействия с реляционными БД, в нем не предусмотрены перечисленные ниже возможности, которые в ряде случаев могут оказаться полезными [1]:

- 1) перехват исключений, возникающих при обработке данных;
- 2) организация итераций по возвращаемым строкам данных;
- 3) защита кода посредством шифрования;
- 4) хранение кода в БД, а не только на клиентской машине;
- 5) организация принятия решений и альтернативного ветвления при обработке строк таблицы.

Указанные возможности реализуются с помощью процедурных языков, которые обычно содержат следующие языковые элементы расширения SQL:

- 1) объявление и инициализацию локальных переменных, оператор присваивания;
- 2) условные операторы (оператор ветвления и оператор цикла);
- 3) курсоры (явный или неявный, в зависимости от конкретной СУБД) для выполнения цикла при просмотре строк;
- 4) генератор последовательности целых значений;
- 5) генерацию сообщения об исключительной ситуации и прерывание выполнения кода триггера или хранимой процедуры;
- 6) оператор для выполнения запросов в блоках, хранимых процедурах и триггерах (в Firebird это EXECUTE STATEMENT, в MS SQL Server — EXECUTE, в Oracle — EXECUTE IMMEDIATE).

Таким образом, в процедурном SQL обычно могут использоваться следующие основные операторы:

- 1) DQL и DML SQL-запросы. Процедурный SQL был разработан для реализации последовательности выполнения SQL-запросов,

с помощью которых происходит взаимодействие с БД. Напрямую с помощью SQL-запросов можно только выбирать данные (SELECT) и производить их изменение (INSERT, DELETE, UPDATE, MERGE);

2) DDL SQL-запросы — напрямую не поддерживаются и могут быть выполнены через динамический SQL;

3) DCL SQL-запросы — напрямую не поддерживаются и могут быть выполнены через динамический SQL;

4) команды управления транзакциями;

5) управляющие операторы: операторы условных переходов, перехода по метке, прерывания выполнения, циклы;

6) курсоры;

7) автоинкремент;

8) операторы перехвата и обработки исключений.

Программа на любом процедурном языке имеет модульную структуру, которую можно представить в виде четырех разделов (блоков).

Первым разделом является *заголовок*. Он используется для присвоения программе имени, а также для задания типа программы (например, функция, процедура или триггер). Если раздел заголовка используется, то это означает, что программа является поименованным блоком и ее можно вызывать по имени.

Вторым разделом является *раздел объявления переменных* (раздел деклараций). В нем указываются имена, типы, значения переменных, которые используются в программе. Если переменные отсутствуют, то этот раздел не используется.

Самый важный и необходимый раздел — *раздел исполнения* (раздел исполняемого кода), в котором непосредственно на процедурном языке описывается вся логика программы. Раздел исполняемого кода — это единственный раздел, который обязательно должен присутствовать в любой программе на процедурном языке, и в нем должно находиться как минимум одно исполняемое выражение.

Последний раздел — необязательный *раздел исключений*, предназначенный для нахождения и обработки всех случаев неверной работы программ.

В отличие от стандартного SQL, процедурные языки в разных СУБД имеют существенные различия, даже для программирования однотипных объектов. Настоящий учебник не претендует на рассмотрение всех особенностей процедурного программирования используемых СУБД, а описывает только базовые принципы.

6.1.1. SQL-сценарии

Для создания БД и ее объектов используется пакет запросов языка определения данных и команд процедурного языка (PSQL, PL/SQL или T-SQL для рассматриваемых СУБД), как правило, оформ-

ленный в виде текстового файла, называемого SQL-сценарием (файл-сценарием, скриптом).

Скрипт — это текстовый файл, содержащий запросы и операторы процедурного языка и обычно имеющий расширение `sql`. Все запросы, команды и операторы SQL-сценария выполняются последовательно в том порядке, в котором они следуют в скрипте.

В общем случае скрипт может содержать запросы создания и модификации всех объектов БД (скрипт DDL), запросы изменения данных (скрипт DML) и команды фиксации/отмены транзакций. Основное предназначение скриптов — автоматическое выполнение административных задач.

Запросы выборки данных в скрипте использовать нельзя, кроме случаев:

1) передачи запроса в строке с помощью специального оператора для выполнения динамических запросов (в зависимости от СУБД — `EXECUTE STATEMENT`, `EXECUTE` либо `EXECUTE IMMEDIATE`, будут подробно рассмотрены позднее);

2) использования запроса для задания набора данных курсора;

3) перенаправления результата запроса `SELECT` во внешний файл командой `OUTPUT` в Firebird.

Для создания скриптов может использоваться любой текстовый редактор, например редактор скриптов в IBExpert при работе с БД Firebird или редактор dbForge Studio при работе с БД MS SQL Server и Oracle Database.

Система управления БД Firebird допускает использование в файле скрипта команды

```
OUTPUT filespec;
```

где `filespec` — спецификация внешнего файла.

Команда `OUTPUT` перенаправляет данные во внешний файл на диске (если `filespec` указана) или на стандартное устройство вывода на монитор (если `filespec` не указана). Например, можно перенаправить данные, возвращаемые запросом `SELECT`, во внешний файл, а потом закрыть этот файл командой `OUTPUT` без параметров:

```
OUTPUT C:\1.txt;
SELECT * FROM Abonent;
OUTPUT;
```

Следует отметить, что указываемый файл может и не существовать на диске и в случае отсутствия будет создан при выполнении команды `OUTPUT`. С помощью команды `OUTPUT` таким образом можно создать и `sql`-файл скрипта [5].

В СУБД MS SQL Server и Oracle команда `OUTPUT` в скриптах не используется, перенаправление данных во внешний файл реализуется с помощью утилит командной строки и других средств [8, 9].

Текстовый файл скрипта может содержать команду на создание БД CREATE DATABASE. При этом важно понимать, что различные СУБД хранят данные совершенно по-разному, в связи с чем имеются существенные различия в синтаксисе команды CREATE DATABASE. Обычно создание БД «вручную», а не с помощью графического инструментария выполняется только квалифицированным администратором БД.

В Firebird команда в простейшем случае имеет синтаксис

```
CREATE DATABASE 'filespec'  
[USER 'user_name' [PASSWORD 'user_password']]  
[PAGE_SIZE [=] int ] [DEFAULT CHARACTER SET charset];
```

где filespec — спецификация файла новой БД; USER 'user_name' — задает имя пользователя (может содержать до 31 символа, нечувствительно к регистру); PASSWORD 'user_password' — задает пароль (может содержать до 64 символов, при этом только первые 20 имеют значение, чувствителен к регистру); PAGE_SIZE = int — устанавливает размер в байтах страниц БД (допустимо 1024, 2048, 4096, 8192 и 16 384); DEFAULT CHARACTER SET charset — устанавливает набор символов с именем charset для БД, используемый по умолчанию. Если установка набора символов опущена, то в качестве него по умолчанию принимается NONE.

При создании БД в MS SQL Server очень важно понимать, как SQL Server хранит данные, чтобы посчитать и указать количество дискового пространства, требуемого для размещения БД. При создании баз необходимо учитывать следующее:

1) все БД имеют основной файл данных, определяемый именем файла с расширением .mdf, и один или более файлов журнала, определяемых именем файла с расширением .ldf;

2) файлы MS SQL Server имеют два имени: физическое имя (имя физического файла, которое должно соответствовать правилам для имен файлов ОС) и логическое имя (имя, используемое для ссылки на физический файл в операторах T-SQL, которое должно соответствовать правилам для идентификаторов SQL Server и быть уникальным в соответствующей БД);

3) когда создается БД, в нее копируется содержимое БД model, которая включает системные таблицы и может содержать пользовательские таблицы. Минимальный размер создаваемой БД должен быть равен или больше размера БД model;

4) в отличие от Firebird, MS SQL Server хранит, читает и записывает данные блоками по 8 килобайт (Кбайт), которые называются страницами. База данных может хранить 128 страниц на мегабайт (Мбайт), так как 1 Мбайт содержит 1024 Кбайта (1024 Кбайта разделить на 8 Кбайт равно 128). Все страницы хранятся в простран-

стве. Пространство — это восемь последовательных страниц, или 64 Кбайта. Поэтому БД имеет 16 пространств в Мбайт.

Примечание. В MS SQL размер страниц всегда равен 8 Кбайт.

В MS SQL Server команда создания БД имеет формат [7]

```
CREATE DATABASE имя_базы
[ON [< filespec > [ ,... ] ] [ , < filegroup > [ ,... ] ]]
[LOG ON { < filespec > [ ,... ] }]
[COLLATE кодировка]
[FOR LOAD | FOR ATTACH];
```

где

```
< filespec > ::= [PRIMARY]
                ( NAME = логическое_имя_файла,
                  FILENAME = 'имя_файла_в_ОС'
                  [, SIZE = размер]
                  [, MAXSIZE = { максимальный_размер | UNLIMITED } ]
                  [, FILEGROWTH = увеличение ] ) [ ,... ] ;
< filegroup > ::= FILEGROUP файловая_группа < filespec > [ ,... ]
```

В отличие от Firebird, в MS SQL Server при создании БД в обязательном порядке задается не спецификация файла БД, а имя_базы, которое должно быть уникально внутри экземпляра SQL Server и соответствовать правилам для идентификаторов.

Секция ON указывает, что дисковые файлы, используемые для хранения разделов данных в БД, определяются явно. Список спецификаций файлов <filespec> определяет файлы данных первичной файловой группы, за ним может следовать необязательный список элементов <filegroup>, которые определяют файловые группы пользователей и принадлежащие им файлы. Первичная файловая группа содержит все системные БД, а также все объекты, не назначенные другим файловым группам.

Опция PRIMARY определяет основной файл в первичной файловой группе. Файловое расширение основного файла — .mdf. Если опция PRIMARY не указана, то первый файл, указанный в списке <filespec> в первичной файловой группе, становится первичным файлом. В БД может быть только один первичный файл.

Параметр NAME задает логическое имя файла, а FILENAME указывает имя и путь к файлу в ОС. Путь должен указывать на папку на сервере, где установлен SQL Server. Нельзя использовать сетевые диски с других компьютеров.

Параметр SIZE указывает размер файла данных или журнала. Размер может быть задан в Мбайт (значение по умолчанию) или в Кбайт.

Параметр MAXSIZE указывает максимальный размер, до которого файл может увеличиваться. Размер может быть задан в Мбайт (значение по умолчанию) или в Кбайт. Если не указать максимальный размер, то файл станет увеличиваться, пока диск не будет заполнен полностью.

Секция LOG ON указывает, что дисковые файлы, используемые для хранения журнала БД, т. е. файлы журналов, определяются явно списком <filespec>. Если секция LOG ON отсутствует, то автоматически создается один файл журнала, размер которого определяется большей из следующих двух величин: 512 Кбайт или 25 % от суммы размеров всех файлов с данными в БД.

Примечание. Если дисковые файлы явно не определяются секциями ON и LOG ON, то SQL Server использует имя_базы для получения логического имени файла и имени файла в ОС. Путь по умолчанию берется из реестра. Его можно изменить в «Свойствах сервера» (страница «Параметры базы данных») в Management Studio. Изменение пути по умолчанию требует перезапуска SQL Server.

Параметр COLLATE задает параметры сортировки по умолчанию для БД. Параметры сортировки (кодировка, сопоставления, раскладка) включают роли, контролирующие использование символов для языка и алфавита. Именем параметров сортировки может быть либо имя параметров сортировки Windows, либо имя параметров сортировки SQL [7]. Если параметр не указан, то базе данных назначаются параметры сортировки по умолчанию для экземпляра SQL Server.

Опция FOR ATTACH указывает, что БД создается путем присоединения существующего набора файлов ОС.

В СУБД Oracle Database новую БД можно создать либо с помощью помощника по конфигурированию — Oracle DBCA (Oracle Database Configuration Assistant), либо вручную командой CREATE DATABASE [10]. При этом следует учесть, что в настоящем учебнике используется бесплатная версия СУБД Oracle Express Edition (XE), которая позволяет работать только с одной БД, создаваемой сразу при установке Oracle. Таким образом, описание создания БД командой CREATE DATABASE приводится с ознакомительной целью, без практического апробирования в Oracle.

Создание БД вручную в Oracle имеет множество существенных особенностей, так как среда Oracle Database является в высокой степени конфигурируемой, и это преимущество накладывает на администраторов БД обязанность знания работы многочисленных параметров и их возможного взаимодействия друг с другом [10]. В настоящем учебнике не рассматриваются особенности конфигурирования БД Oracle, отмечены лишь основные моменты.

Прежде чем приступить к созданию БД Oracle, необходимо создать файл параметров инициализации БД (initИмя_базы.ora). За исключением DB_NAME, определяющего имя БД, все остальные параметры инициализации Oracle являются необязательными (если не заданы, используются значения по умолчанию).

После того как файл инициализации создан, экземпляр Oracle можно запускать, используя этот файл в командах запуска БД. Как только экземпляр запущен, можно создать соответствующую БД командой следующего формата:

```
CREATE [CONTROLFILE REUSE] DATABASE имя_базы
[USER SYS IDENTIFIED BY 'sys_password']
[USER SYSTEM IDENTIFIED BY 'system_password']
{[LOGFILE GROUP номер_группы ('filespec') SIZE размер [ ,... ]]
[MAXLOGFILES число]
[MAXLOGMEMBERS число]
[MAXLOGHISTORY число]
[MAXDATAFILES число] }
[CHARACTER SET charset ]
[NATIONAL CHARACTER SET charset ]
[EXTENT MANAGEMENT LOCAL
DATAFILE 'filespec' SIZE размер [REUSE]]
[SYSAUX DATAFILE 'filespec' SIZE размер [REUSE]]
[DEFAULT TABLESPACE имя_пространства
DATAFILE 'filespec' SIZE размер [REUSE]
[AUTOEXTEND ON MAXSIZE UNLIMITED]]
[DEFAULT TEMPORARY TABLESPACE имя_пространства
TEMPFILE 'filespec' SIZE размер [REUSE]]
[UNDO TABLESPACE имя_пространства
DATAFILE 'filespec' SIZE размер [REUSE]
[AUTOEXTEND ON MAXSIZE UNLIMITED]];
```

Команда CREATE DATABASE иницирует создание управляющих файлов, в которых содержится информация, связанная с физическим местоположением каждого файла данных и файла журнала БД в БД. Расположение файлов считывается из файла параметров initИмя_базы.ora, и если управляющие файлы с такими именами уже существуют во время создания БД (остались от предыдущей неудачной инсталляции), необходимо указать опцию CONTROLFILE REUSE.

Необязательные секции USER демонстрируют возможный способ указания паролей для двух ключевых пользователей Oracle — SYS и SYSTEM. Если секции не указаны, то пользователям SYS и SYSTEM по умолчанию присваиваются пароли change_on_install и manager соответственно. Поскольку эти пароли общеизвестны, компания Oracle советует использовать секции USER для изменения паролей по умолчанию.

Секция LOGFILE создает группы оперативных журналов повторного выполнения, необходимых для работы Oracle. Пара-

метры MAXLOGFILES, MAXLOGMEMBERS, MAXLOGHISTORY и MAXDATAFILES задают максимальное количество файлов журналов, файлов данных и т. п.

В секциях CHARACTER SET и NATIONAL CHARACTER SET указываются наборы символов, используемые БД.

Для хранения словаря данных в Oracle применяется табличное пространство System. Опция EXTENT MANAGEMENT LOCAL указывает, что управление табличным пространством System должно осуществляться локально, а не посредством словаря. Секция DATAFILE 'filespec' SIZE размер создает табличное пространство System с указанным файлом данных заданного размера. Словарь данных создается внутри табличного пространства System. Один сегмент отката системы также создается автоматически. Опция REUSE указывает на замену файла при его существовании.

Секция SYSAUX DATAFILE 'filespec' SIZE размер создает новое используемое по умолчанию табличное пространство Sysaux с указанным файлом заданного размера.

Секция DEFAULT TABLESPACE создает заданное по умолчанию постоянное табличное пространство с указанным файлом заданного размера.

Секция DEFAULT TEMPORARY TABLESPACE создает новое заданное по умолчанию временное табличное пространство БД. Любым пользователям, которым постоянное табличное пространство не присвоено явно, это табличное пространство будет выделено в качестве используемого по умолчанию, вместо табличного пространства System. Файлом, используемым для временного табличного пространства, является временный (temp) файл, а не обычный файл данных. Временное табличное пространство нельзя создать, используя спецификацию обычного файла данных.

Секция UNDO TABLESPACE создает для новой БД новое табличное пространство отката.

Создание БД в Oracle может быть довольно сложным процессом. При этом существует простейший способ быстрого создания новой базы, который состоит из следующих шагов [10]:

- 1) создание нового файла initИмя_базы.ora только с одним параметром DB_NAME (например, DB_NAME=Abonent);
- 2) запуск нового экземпляра Oracle из интерфейса SQL*Plus с помощью команды STARTUP NOMOUNT;
- 3) создание БД командой без каких-либо параметров:

```
CREATE DATABASE;
```

Oracle автоматически создаст табличные пространства System и Sysaux [11]. База данных будет запущена в режиме ручного управления откатом с использованием сегментов отката.

При создании БД в любой СУБД существенное значение имеет задание *кодовой таблицы* (набора символов), используемой по умолчанию для данных, хранимых в базе. Для хранения данных на русском языке пригодны два варианта.

1. *Создание БД без задания кодовой таблицы.* Символьные данные хранятся в базе в том виде, как они были загружены, без каких-либо предварительных преобразований. Сортировка данных осуществляется в порядке возрастания кодов хранимых символов.

Команда на создание БД без задания кодовой таблицы в Firebird имеет синтаксис

```
CREATE DATABASE 'filespec'  
[USER 'user_name' [PASSWORD 'user_password']]  
[PAGE_SIZE [=] int ] [DEFAULT CHARACTER SET NONE];
```

в MS SQL Server будет иметь вид

```
CREATE DATABASE имя_базы  
[ ON [< filespec > [ ,... ] ] [ , < filegroup > [ ,... ] ] ]  
[ LOG ON { < filespec > [ ,... ] } ];
```

а в Oracle не будет содержать секций CHARACTER SET и NATIONAL CHARACTER SET.

Например, для создания БД без задания кодовой таблицы можно использовать в Firebird команду

```
CREATE DATABASE 'c:\sqllab.fdb'  
USER 'SYSDBA' PASSWORD 'masterkey'  
PAGE_SIZE 16384  
DEFAULT CHARACTER SET NONE;
```

а в MS SQL Server команду

```
CREATE DATABASE sqllab  
ON (NAME = N'SYSDBA', FILENAME = N'C:\SQLDATA\sqllab.MDF', SIZE = 10);
```

2. *Создание БД с кодовой таблицей русского языка.* В каждой СУБД используется своя кодировка для русского языка и символов кириллицы. В Firebird, например, это кодировка WIN1251, и в этом случае команда на создание БД имеет синтаксис

```
CREATE DATABASE 'filespec'  
[USER 'user_name' [PASSWORD 'user_password']]  
[PAGE_SIZE [=] int ] [DEFAULT CHARACTER SET WIN1251];
```

В MS SQL Server команда должна быть записана в следующем виде:

```
CREATE DATABASE 'filespec'  
[ ON [< filespec > [ ,... ] ] [ , < filegroup > [ ,... ] ] ]
```

```
[ LOG ON { < filespec > [ ,... ] } ]  
COLLATE cyrillic_general_ci_as;
```

Чтобы получить полный список всех используемых в MS SQL Server схем сопоставления (кодировок) с расшифровкой, можно выполнить запрос

```
SELECT * FROM ::fn_helpcollations(); .
```

Oracle предлагает рекомендованное решение в виде кодировки AL32UTF8:

```
CREATE DATABASE имя_базы  
CHARACTER SET AL32UTF8;
```

Во многих СУБД для уже созданной БД кодовую страницу можно поменять с помощью команды ALTER DATABASE. В Firebird, например, такое изменение задается в виде

```
ALTER DATABASE 'filespec'  
ALTER DEFAULT CHARACTER SET charset;
```

а в MS SQL Server

```
ALTER DATABASE 'filespec'  
COLLATE имя_раскладки;
```

Следует отметить, что такое изменение кодовой страницы не затрагивает существующих данных, а только устанавливает набор символов по умолчанию при внесении новых данных.

В СУБД Oracle смена кодировки для уже созданной БД — довольно сложный процесс, требующий выполнения не только команды ALTER DATABASE, но и ряда дополнительных действий [12]. При этом версия Express Edition 11g вообще не поддерживает смену кодировки единственной рабочей БД, она всегда AL32UTF8.

Для физического удаления БД используется команда DROP DATABASE. В Firebird команда используется для удаления подключенной БД и не имеет параметров:

```
DROP DATABASE;
```

В MS SQL Server БД (или моментальный снимок БД) может быть удалена командой следующего формата:

```
DROP DATABASE { имя_базы | имя_снимка_базы } [ ,... ];
```

При удалении БД она удаляется из экземпляра SQL Server, а ее физические файлы удаляются с диска. Если БД или один из ее файлов

во время удаления находится в режиме вне сети, то файлы с диска не удаляются. Возможно удаление нескольких БД одной командой. Следует помнить, что невозможно удалить системные базы, а также удалить БД, которая используется в текущий момент времени.

В Oracle команда удаления БД имеет формат

```
DROP DATABASE [INCLUDING BACKUPS] [NOPROMPT];
```

Опция INCLUDING BACKUPS удаляет резервные и прокси-копии, снимки БД и архивные журналы, связанные с целевой БД, из всех настроенных типов устройств. Опция NOPROMPT позволяет не запрашивать подтверждение перед удалением БД.

Как и в случае с командой ALTER DATABASE, удаление БД в Oracle требует ряда дополнительных действий помимо команды DROP DATABASE. Попытка выполнения сразу команды DROP DATABASE вызывает ошибку вида «ORA-01586: database must be mounted EXCLUSIVE and not open for this operation». Подробную информацию по удалению, а также по созданию и изменению БД Oracle можно найти в официальных руководствах [11].

Если используется СУБД Firebird, то важной характеристикой БД, помимо размера страницы и набора символов, является ее диалект. Создание новой БД в Firebird возможно в диалекте 1 или диалекте 3, однако рекомендуется создавать новые базы в диалекте 3, так как он предоставляет наиболее полный набор возможностей Firebird. Учебная БД, используемая в настоящем учебнике, создана на диалекте 3.

Чтобы задать диалект создаваемой БД в скрипте, следует перед командой CREATE DATABASE выполнить команду

```
SET SQL DIALECT номер_диалекта;
```

где номер_диалекта равен 1 или 3.

Примечание. При создании БД с использованием IBExpert диалект, так же как размер страницы и набор символов, выбирается из выпадающего списка в окне «Создание базы данных».

Скрипт может содержать подключение к БД. В Firebird подключение к БД производится с помощью команды CONNECT:

```
CONNECT 'filespec'  
[USER 'user_name'] [PASSWORD 'user_password'] [ROLE 'user_role']; .
```

Соединиться с имеющейся БД может любой пользователь, зарегистрированный в системе. Например, для подключения администратора к созданной БД c:\sqlab.fdb нужно использовать команду

```
CONNECT 'c:\sqllab.fdb' USER 'SYSDBA' PASSWORD 'masterkey';
```

Для автоматического помещения данной команды в начало создаваемого скрипта достаточно в окне редактора скриптов IBExpert выбрать в меню «Скрипт» пункт «Добавить оператор CONNECT в начало скрипта». В появившемся окне нужно выбрать БД, для которой будет использоваться скрипт.

Скрипт может содержать команду DISCONNECT для принудительного разрыва соединения с БД после выполнения необходимых операций. Команда DISCONNECT имеет синтаксис

```
DISCONNECT 'filespec';
```

Например, для отсоединения от учебной БД с именем sqllab.fdb можно использовать команду

```
DISCONNECT 'c:\sqllab.fdb';
```

В MS SQL Server команд CONNECT и DISCONNECT нет, для подключения к БД используются хранимые процедуры sp_attach_db и sp_detach_db. Например, подключение к БД можно выполнить путем вызова процедуры sp_attach_db в виде

```
EXEC sp_attach_db @dbname = N'sqllab',  
    @filename1 = N'c:\Program Files\Microsoft SQL Server\MSSQL\Data\  
sqllab.mdf',  
    @filename2 = N'c:\Program Files\Microsoft SQL Server\MSSQL\Data\  
sqllab_log.ldf';
```

Для отключения базы можно использовать процедуру sp_detach_db:

```
EXEC sp_detach_db 'sqllab', 'true';
```

Система управления БД Oracle для подключения к БД поддерживает, как и Firebird, команду CONNECT. Команда CONNECT реализуется в формате

```
CONNECT [ { user_name[/password] [@connect_identifer] | / }  
[AS {SYSDBA/SYSOPER} ] ];
```

Задаются параметры авторизации либо используется косая черта (/). Необязательный параметр @connect_identifer задает идентификатор соединения, синтаксис зависит от конфигурации Oracle [11]. Если параметры авторизации не заданы, а указан слэш, подключение выполняется с использованием учетной записи ОС.

Опции AS SYSDBA и AS SYSOPER позволяют устанавливать привилегированные подключения пользователям, которым были вы-

даны системные привилегии SYSDBA или SYSOPER. Определенные операции, например запуск и завершение работы, разрешено выполнять только в случае подключения с привилегиями системных пользователей Oracle SYSDBA или SYSOPER.

Команду DISCONNECT для разрыва существующего подключения к БД платформа Oracle позволяет использовать только в своем инструменте SQL*Plus. Команда вызывается без параметров.

Запуск на выполнение SQL-сценария в различных СУБД может осуществляться с помощью графических утилит либо путем выполнения определенных команд.

При работе с БД Firebird, например в IBExpert, запуск на выполнение SQL-сценария производится из меню редактора скриптов (Ctrl + F12) выбором пункта «Выполнить скрипт» (F9) для уже загруженного скрипта и пункта «Run from file...» (Shift + F9) для скрипта, находящегося во внешнем файле. Для выполнения скрипта, находящегося во внешнем файле, в Firebird используется также команда

```
INPUT filespec;
```

Следует отметить, что внешний файл может содержать другую команду INPUT, предоставляя, таким образом, возможность выполнения определенной цепочки скриптов друг за другом.

При работе с БД MS SQL Server или Oracle в dbForge Studio, например, для запуска скрипта на выполнение используются следующие «горячие» клавиши:

- 1) Ctrl + N — новое окно скрипта;
- 2) Ctrl + O — открыть файл со скриптом;
- 3) F5 — компиляция и выполнение скрипта.

Операторы процедурного языка, определенные для хранимых процедур и триггеров, также могут включаться в скрипты. Поэтому создание триггеров и хранимых процедур осуществляется с использованием SQL-сценариев. При выполнении скрипта происходит компиляция хранимых процедур и триггеров, включенных в текст данного SQL-скрипта.

В СУБД Firebird блоки PSQL получают специальную трактовку в скриптах в зависимости от символов терминатора, разделяющего друг от друга операторы (точка с запятой или тот, который задан с помощью оператора SET TERM) [5].

В файле SQL-сценария рекомендуется использовать разные терминаторы для разделения внутренних операторов ХП или триггера и внешних запросов (команд). Новый внешний терминатор в Firebird определяется оператором SET TERM в формате

```
SET TERM новый_терминатор старый_терминатор;
```

Альтернативный терминатор может быть любой строкой символов, за исключением зарезервированного ключевого слова, точки

с запятой, пробела или апострофа [5]. Например, чтобы определить в качестве терминатора два восклицательных знака (!!) вместо обычного разделителя (;), необходимо использовать оператор

```
SET TERM  !!  ;
```

После этого точка с запятой будет использоваться для разделения внутренних операторов ХП или триггера, а два восклицательных знака (!!) — для отделения ХП, триггеров и других запросов (команд) друг от друга в файле SQL-сценария. После использования нового терминатора старый можно восстановить командой

```
SET TERM ; !!
```

В MS SQL Server аналогом понятия PSQL-блоков Firebird являются пакеты. Пакеты разделяются между собой операторами GO. Следует иметь в виду, что область видимости переменных ограничивается одним пакетом и каждый пакет обособлен от других.

В СУБД Oracle принято в файл скрипта с кодом хранимой процедуры включать косую черту (/) после запроса на создание процедуры. В таком случае не приходится специально заботиться о компиляции процедуры — система автоматически компилирует процедуру и сохраняет ее в случае отсутствия ошибок [1]. Косая черта должна быть указана отдельной строкой, в которой нет никаких других операторов и команд.

В любом месте программы может быть поставлена метка, имеющая в Firebird, MS SQL Server синтаксис

```
имя_метки:
```

и в Oracle синтаксис

```
<<[имя_метки]>>
```

где *имя_метки* — произвольный идентификатор, позволяющий именовать некоторый оператор модуля и таким образом ссылаться на него. Допускается в качестве меток использование целых чисел без знака. Метка располагается перед оператором на той же или отдельной строке. Метки следует использовать для структурирования программы и более гибкого управления выходом из вложенных циклов.

В запросах SQL, операторах и скриптах можно использовать комментарии. *Комментарий* — это произвольный текст, предназначенный для пояснения работы отдельных частей программы (запроса) или содержать любую другую полезную информацию. На выполнение команд SQL комментарии не влияют, так как синтаксический

анализатор игнорирует текст комментариев. Однако код без комментариев — это непрофессионально написанный код.

В Firebird поддерживаются два типа комментариев:

1) *блочный комментарий*, заданный ограничителями `/*` и `*/`, содержит одну или несколько строк комментария. Как правило, располагается на отдельной строке (или строках), но может располагаться и в той же строке, что и запрос SQL, причем в любом месте запроса SQL. Например:

```
/* Это  
SELECT * FROM Abonent WHERE 1 = 1;  
комментарий */ ;
```

или

```
SELECT * FROM Abonent WHERE /* Это комментарий */ 1 = 1;
```

2) *однострочный комментарий* располагается в конце строки кода в скрипте, запросе, хранимой процедуре, триггере. Отделяется от основного текста сценария с помощью двух дефисов, после которых все символы до конца строки игнорируются, например

```
SELECT * FROM Abonent WHERE 1 = 1 -- Это комментарий
```

Однострочный комментарий часто используется для комментирования нежелательных частей утверждений, например

```
... WHERE c1 = 9 OR c2 = 99 -- OR c3 = 999 ...
```

В данном случае часть условия поиска (`OR c3 = 999`) закомментирована. При использовании данного вида комментария игнорируется текст до следующей строки.

В Oracle для комментирования может использоваться также команда `REMARK` (сокращенно `REM`). Команда фактически аналогична однострочному комментарию, добавляется в начале строки и означает, что данную строку выполнять не нужно.

Примечания

1. Рекомендуется создавать комментарии, используя латинские символы, так как извлечение комментариев с кириллицей может вызывать ошибки.
2. Не допускается смешивание однострочного комментария с блочным комментарием.

Чтобы код хорошо читался и был удобен при последующей поддержке, необходимо придерживаться некоторого свода правил

по оформлению кода [13]. Для автоматизации форматирования кода в средах разработки, а также в некоторых графических инструментах для работы с БД есть специальные средства, позволяющие отформатировать весь файл или только выделенную часть в соответствии с установленными правилами форматирования. Например, в dbForge Studio for Oracle текст SQL-сценария можно отформатировать выбором пункта «Форматировать документ» (Ctrl + K, D) на панели инструментов «Текст». В средах разработки обычно правила форматирования можно настроить, чтобы они максимально соответствовали принятому стилю оформления кода. После настройки правил можно быстро форматировать код, значительно экономя силы и время.

Примеры SQL-сценариев создания учебной БД для каждой из СУБД и заполнению ее данными приведены в приложении В, которое можно скопировать со страницы настоящего учебника на сайте <https://rgrty.ru>.

6.1.2. Переменные

В модулях на процедурном языке могут использоваться переменные, необходимые для работы модулей. Переменные используются для следующих целей [5]:

- 1) временное хранение данных;
- 2) манипуляция хранимыми значениями;
- 3) возможность многократного использования;
- 4) простота обслуживания;
- 5) повышение читаемости программы.

Всего, как правило, используется четыре типа переменных, причем возможность применения конкретной переменной определяется тем, является ли модуль хранимой процедурой или триггером. Различают локальные переменные, контекстные переменные, входные параметры и выходные параметры.

Локальные переменные используются для хранения значений как в ХП и в триггерах, так и в неименованных блоках процедурного языка. Подобно переменным в других языках программирования, переменная процедурного языка резервирует память для хранения значений, используемых исключительно в рамках конкретного модуля. За пределами модуля переменные недоступны, поэтому они по сути являются *закрытыми переменными* [1].

Следует отметить, что в некоторых СУБД (например, в MS SQL Server и Oracle) в скриптах могут использоваться *глобальные переменные*, к которым возможно обращение из-за пределов блока процедурного языка. Также следует отметить, что в MS SQL Server переменные можно использовать не только в модулях на процедурном языке, но и в T-SQL.

Объявление локальной переменной в теле хранимой процедуры, триггера или неименованного блока осуществляется в *разделе деклараций* с помощью оператора DECLARE.

В Firebird локальные переменные объявляются в следующем формате:

```
DECLARE [VARIABLE] имя_локальной_переменной <тип_переменной>
    [{ = | DEFAULT} <значение>];
```

где

```
<тип_переменной> ::= {<тип_данных> | имя_домена | TYPE OF имя_домена
    | TYPE OF COLUMN {таблица | представление}.столбец
}
```

Для каждой переменной используется свой оператор DECLARE, который заканчивается точкой с запятой. Правило составления имен локальных переменных такое же, как и для любых других идентификаторов языка SQL (см. параграф 2.7).

Синтаксис объявления переменных в MS SQL Server имеет вид

```
DECLARE { {@имя_локальной_переменной
    [AS] <тип_переменной> [ = <значение>]} [,...]
    | @имя_табличной_переменной
    [AS] TABLE ( { <определение_столбца>
    | <ограничение_таблицы> } [,...] ) };
```

где

```
<тип_переменной> ::= {<тип_данных> | имя_домена | AS имя_домена }
```

а имя_домена фактически представляет собой пользовательский тип данных, определенный с помощью запроса CREATE TYPE.

Имена переменных в MS SQL Server начинаются с символа @, но не следует использовать два символа @@, потому что так начинаются некоторые системные функции. Помимо переменных базовых типов могут использоваться локальные переменные табличного типа (TABLE). Такие переменные используются, в первую очередь, для временного хранения набора строк, возвращаемых как результирующий набор функции с табличным значением. Один оператор DECLARE позволяет объявить несколько локальных переменных одинакового или разного типов через запятую.

В Oracle правила объявления переменных несколько различаются для именованных (хранимых процедур, функций, триггеров) и неименованных блоков. Общий формат объявления переменных такой [1]:

```
[ DECLARE ] {имя_переменной [CONSTANT] <тип_переменной> [NOT NULL]
    [{ := | DEFAULT} <значение>] } [;..]
```

где

```
<тип_переменной> ::= {<тип_данных> | таблица.столбец %TYPE }
```

При этом оператор DECLARE в Oracle используется только в неименованных блоках PL/SQL. Одним оператором DECLARE может быть объявлено несколько переменных или констант (через точку с запятой). Объявления в разделе деклараций именованных модулей Oracle не используют оператор DECLARE.

Константа внешне похожа на переменную, но ее значение не может изменяться в процессе выполнения программы. Применять именованные константы вместо конкретных значений удобно, так как при этом можно изменить значения в блоке, модифицировав объявление в разделе деклараций. Даже если значение в блоке встречается только один раз, желательно объявить его как именованную константу. Это делает программу более удобной для восприятия.

Имя переменной в Oracle может содержать до 30 символов (букв, цифр и специальных знаков) и должно начинаться с буквы. Не допускается совпадение имени переменной с именем столбца, ссылка на который содержится в блоке PL/SQL. Многие программисты Oracle придерживаются соглашения об именовании, согласно которому имена констант, переменных и глобальных переменных начинаются соответственно с *c_*, *v_* или *g_*. Соглашение не является обязательным, но делает программу удобной для восприятия.

Опция NOT NULL указывает, что переменная должна быть обязательно инициализирована, в противном случае Oracle сгенерирует исключение.

В рассматриваемых СУБД в качестве типа данных переменной можно указывать:

1) в любой из СУБД — один из системных типов (см. параграф 2.8). В Oracle дополнительно к системным типам для переменных PL/SQL может использоваться тип Boolean;

2) в Firebird и MS SQL Server — ранее созданный домен или конструкцию, которая извлекает только тип из существующего домена (ограничения и значения по умолчанию, определенные для данного домена, не учитываются). Для Firebird это конструкция *TYPE OF <имя_домена>*, для MS SQL Server — *AS <имя_домена>*;

3) в Firebird и Oracle — конструкцию для заимствования типа данных столбца таблицы (или представления). В Firebird это конструкция вида *TYPE OF COLUMN {таблица | представление}.столбец*, а в Oracle — конструкция *таблица.столбец%TYPE*.

Например, чтобы объявить переменную *x* целого типа, в Firebird можно использовать оператор

```
DECLARE VARIABLE x INTEGER;
```

В MS SQL Server такая переменная должна быть объявлена в виде

```
DECLARE @x INTEGER;
```

а в Oracle —

```
DECLARE x INTEGER;
```

Объявить переменную у на существующем домене в Firebird можно следующим образом:

```
DECLARE VARIABLE y Money;
```

а в MS SQL Server в виде

```
DECLARE @y Cash;
```

Объявить переменную New_Phone с типом данных, совпадающим с типом столбца Phone таблицы Abonent, в Firebird можно так:

```
DECLARE VARIABLE New_Phone TYPE OF COLUMN Abonent.Phone;
```

а в Oracle так:

```
DECLARE New_Phone Abonent.Phone%TYPE;
```

Объявить переменную табличного типа MyTableVar в MS SQL Server можно следующим образом:

```
DECLARE @MytableVar TABLE (AccountCD VARCHAR(6) PRIMARY KEY, Fio CHAR(20));
```

Объявить сразу несколько переменных одним оператором DECLARE в MS SQL Server можно так:

```
DECLARE @FirstName NVARCHAR(20), @StateProvince NCHAR (2);
```

В Oracle аналогичное объявление будет выглядеть как

```
DECLARE
  FirstName VARCHAR2(20);
  StateProvince CHAR (2);
```

Все локальные переменные должны быть объявлены до первого выполняемого блока процедурного языка (триггера, процедуры).

Допускается при объявлении локальной переменной сразу же ее инициализировать. Например, в Firebird:

```
DECLARE VARIABLE x INTEGER = 123;
```

ИЛИ

```
DECLARE VARIABLE x INTEGER DEFAULT 123;
```

В MS SQL Server:

```
DECLARE @x INTEGER = 123;
```

В Oracle значение инициализируется так:

```
DECLARE x INTEGER := 123;
```

или так:

```
DECLARE x INTEGER DEFAULT 123;
```

Константа в Oracle объявляется следующим образом:

```
DECLARE x CONSTANT INTEGER := 123;
```

Объявленным переменным в дальнейшем могут присваиваться различные значения с помощью оператора присваивания. Переменным должны присваиваться значения того типа данных, с каким они были объявлены.

В Firebird оператор имеет синтаксис

```
имя_локальной_переменной = <выражение>;
```

В Oracle синтаксис оператора присваивания следующий:

```
имя_локальной_переменной := <выражение>;
```

В MS SQL Server присваивание выполняется с помощью оператора SET:

```
SET имя_локальной_переменной = <выражение>;
```

Этот способ присваивания значений переменным является предпочтительным в MS SQL Server. В качестве присваиваемого значения выражения может быть указан скалярный запрос SELECT. Кроме того, переменной можно присвоить значение, указав ее в списке выбора скалярного запроса SELECT.

Например, присваивание значения ранее инициализированной переменной в Firebird может выглядеть так:

```
New_Phone = '556601';
```

в Oracle так:

```
New_Phone := '556601';
```

а в MS SQL Server:

```
SET @New_Phone = '556601';
```

Следующие примеры демонстрируют использование запроса SELECT для присваивания значения переменной в MS SQL Server. Предполагается, что объявлена переменная MaxPay, и ей присваивается максимальное значение оплаты из таблицы PaySumma. Запрос SELECT может быть указан в качестве присваиваемого значения в операторе SET:

```
SET @MaxPay = (SELECT MAX(PaySum) FROM PaySumma);
```

либо может быть указана переменная в списке выбора запроса без использования оператора присваивания:

```
SELECT @MaxPay = MAX(PaySum)  
FROM PaySumma;
```

Примером использования табличной переменной в MS SQL Server может служить запрос на заполнение переменной и последующий вывод данных:

```
INSERT INTO @MytableVar SELECT AccountCD, Fio FROM Abonent;  
SELECT * FROM @MytableVar;
```

Любая СУБД делает доступным множество переменных, поддерживаемых в контексте текущего соединения клиента и его деятельности. Эти *контекстные переменные* доступны для использования в процедурных языках.

В Firebird многие **контекстные переменные** процедурного языка используются только в триггерах. Как в триггерах, так и в ХП может использоваться переменная ROW_COUNT, которая возвращает количество строк, полученных выполненным запросом DML или SELECT. Также и в триггерах, и в ХП могут использоваться контекстные переменные даты и времени и текущего соединения (см. параграф 2.8).

В триггерах для реализации отслеживания целостности данных используются контекстные переменные OLD.столбец и NEW.столбец. Эти переменные хранят старые (до изменения) и новые (после изменения) значения указанного столбца таблицы, когда выполняется запрос на модификацию данных.

Также в триггерах могут использоваться логические контекстные переменные UPDATING, INSERTING и DELETING для задания определенных действий в зависимости от типа события DML, для которого срабатывает триггер.

Входные параметры недоступны для использования в триггерах. Они применяются для передачи значений хранимым процедурам из клиентских приложений или других хранимых процедур.

Выходные параметры также недоступны для использования в триггерах. Они используются для возвращения значений из ХП вызвавшим их объектам.

В MS SQL Server системные контекстные переменные начинаются с символов @@ и фактически представляют собой системные функции. Например, число строк, затронутых при выполнении последней инструкции, возвращает функция @@ROWCOUNT, являющаяся аналогом контекстной переменной ROW_COUNT в Firebird.

Предпочтительным способом для получения контекстных данных по текущему сеансу в MS SQL Server является использование переменной CONTEXT_INFO. Переменная сессии CONTEXT_INFO позволяет сохранять данные переменной длины размером до 128 байт. Чтобы установить значение переменной сессии, нужно воспользоваться командой SET CONTEXT_INFO, а чтобы получить текущее значение, — функцией CONTEXT_INFO(). Например, можно объявить переменную BinVar и установить ее значение, а затем сделать ее доступной в контексте сессии:

```
DECLARE @BinVar varbinary(128);
SET @BinVar = CAST(REPLICATE(0x20, 128) AS varbinary(128));
SET CONTEXT_INFO @BinVar;
SELECT CONTEXT_INFO() AS MyContextInfo;
```

В Oracle, как и в предыдущих СУБД, число строк, полученных последним выполненным запросом DML или SELECT, доступно через контекстную переменную — SQL %ROWCOUNT. Наборы переменных среды доступны через определенные внутренние пространства имен контекста приложения [10]. Для получения информации о состоянии среды окружения используется функция SYS_CONTEXT со следующим синтаксисом [2, 14]:

```
SYS_CONTEXT( пространство_имен, параметр, [длина] )
```

Если необходимо получить атрибуты описания текущей сессии Oracle, используется встроенное пространство имен USERENV.

Каждое пространство имен содержит набор предопределенных параметров. Пространство имен USERENV содержит такие важные параметры, как CURRENT_USER (имя пользователя, запустившего сеанс), DB_NAME (имя базы данных), HOST (имя компьютера, на котором действует БД), OS_USER (имя учетной записи в операционной системе) и др. [2].

Указание длины возвращаемого значения не является обязательным, по умолчанию это 256 байт.

Например, получить имя учетной записи ОС, применяемое пользователем БД Oracle, можно следующим образом:

```
SELECT SYS_CONTEXT ('USERENV', 'OS_USER') FROM DUAL;
```

Говоря о переменных процедурного языка в СУБД Firebird, следует особо отметить то, что в запросах SQL локальные переменные, входные и выходные параметры должны использоваться с *двоеточием перед именем*, чтобы указать, что это переменные, а не имена столбцов таблиц. В остальных случаях (например, при присваивании значения переменной, в логических условиях условных операторов) не требуется наличия двоеточия перед именем переменной. *Двоеточие перед контекстными переменными не ставится никогда.*

Примеры практического использования всех четырех типов переменных в модулях на процедурных языках различных СУБД приведены при изучении ХП и триггеров.

6.1.3. Операторы ветвления, циклов и переходов

При программировании на процедурных языках существует возможность использовать:

- ветвление;
- циклическое выполнение группы операторов;
- переходы.

6.1.3.1. Условный оператор

Для осуществления ветвления по условию в процедурных языках, как и в большинстве языков программирования, используется оператор IF. Оператор применяется во всех СУБД, но имеет несколько различающийся формат реализации. В СУБД Firebird оператор ветвления IF представлен следующим синтаксисом:

```
IF (логическое_условие) THEN <группа_операторов1>  
[ELSE <группа_операторов2>]
```

где логическое_условие — любой предикат, который должен быть истинным, чтобы выполнялась <группа_операторов>, следующая за THEN;

```
<группа_операторов> ::= { <блок_операторов>  
                        | DML_оператор  
                        | оператор_процедурного_языка };  
  
<блок_операторов> ::=  
BEGIN  
    [<группа_операторов> ...]  
END
```

Блок операторов — это группа операторов SQL, рассматриваемых как единое целое. Блок операторов начинается с зарезервиро-

ванного слова BEGIN и заканчивается зарезервированным словом END. При этом блок операторов может содержать вложенный блок операторов, заключенный в операторные скобки BEGIN END, и т. д. Для вложенных блоков после зарезервированного слова END разделительная точка с запятой не ставится. Следует отметить, что между BEGIN...END может не быть ни одного оператора, т. е. допустим «пустой» блок операторов.

Следует отметить, что если <группа_операторов> представляет собой один-единственный DML_оператор или оператор_процедурного_языка, то заключать данный оператор в скобки BEGIN ... END не обязательно.

В MS SQL Server формат оператора IF отличается отсутствием ключевого слова THEN:

```
IF [( ) логическое_условие ( )] <группа_операторов1>  
    [ELSE <группа_операторов2>];
```

Для этого и других операторов ветвления и циклов в MS SQL Server логическое условие должно быть обязательно заключено в скобки только в том случае, если оно представляет собой запрос SELECT.

В Oracle оператор IF имеет расширенный формат и позволяет реализовать логику со многими альтернативными действиями:

```
IF логическое_условие1 THEN <группа_операторов1>  
    [ELSIF логическое_условие2 THEN <группа_операторов2>]  
    [;...]  
    [ELSE <группа_операторов>]  
END IF;
```

Конструкция IF...ELSIF... фактически представляет собой один из способов реализации функций оператора CASE. Условия в конструкции IF...ELSIF... всегда выполняются в порядке их следования в операторе.

Следующий фрагмент кода иллюстрирует использование оператора IF в теле ХП Firebird в предположении, что Street_Cod, House_Nom, Flat_Nom были ранее объявлены как локальные переменные или входные параметры, а Line — как локальная переменная или выходной параметр:

```
IF (House_Nom IS NOT NULL AND Flat_Nom IS NOT NULL) THEN  
    Line = 'Код улицы – ' || Street_Cod || ', д.' || House_Nom || ', кв.'  
        || Flat_Nom;  
ELSE  
    Line = Street_Cod;
```

В MS SQL Server аналогичный код должен выглядеть так:

```

IF (@House_Nom IS NOT NULL AND @Flat_Nom IS NOT NULL)
    SET @Line = 'Код улицы – ' + @Street_Cod + ', д.' + @House_Nom + ',
    кв.'
                + @Flat_Nom;
ELSE
    SET @Line = @Street_Cod;

```

В Oracle может быть организовано множественное ветвление, например определение уровня оплаты v_PayLevel в зависимости от суммы оплаты (переменная v_PaySum):

```

IF v_PaySum <= 50
    THEN v_PayLevel := 'Low Pay';
ELSIF v_PaySum > 150 AND v_PaySum <= 300
    THEN v_PayLevel := 'Avg Pay';
ELSIF v_PaySum > 300 AND v_PaySum <= 500
    THEN v_PayLevel := 'Moderate Pay';
ELSE
    v_PayLevel := 'High Pay';
END IF;

```

В конце каждого оператора ставится точка с запятой, в том числе и перед ELSE и ELSIF.

6.1.3.2. Оператор множественного выбора CASE

Системы управления БД MS SQL Server и Oracle поддерживают использование в процедурном языке оператора множественного выбора CASE.

Как и в обычном SQL, в T-SQL и PL/SQL поддерживаются две формы оператора CASE: простая и с поиском. В T-SQL синтаксис простой формы:

```

CASE выражение
    WHEN значение1 THEN <группа_операторов1>
    [WHEN значение2 THEN <группа_операторов2>]...
    [ELSE <группа_операторов>]
END;

```

Синтаксис формы с поиском:

```

CASE
    WHEN логическое_условие1 THEN <группа_операторов1>
    [WHEN логическое_условие2 THEN <группа_операторов2>]...
    [ELSE <группа_операторов>]
END;

```

В конце каждого оператора, как и в случае с IF, ставится точка с запятой.

В PL/SQL формат простой формы:

```

[<<имя_метки>>]
CASE выражение
    WHEN значение1 THEN <группа_операторов1>
    [WHEN значение2 THEN <группа_операторов2>]...
    [ELSE <группа_операторов>]
END CASE [имя_метки];

```

Синтаксис формы с поиском:

```

[<<имя_метки>>]
CASE
    WHEN логическое_условие1 THEN <группа_операторов1>
    [WHEN логическое_условие2 THEN <группа_операторов2>]...
    [ELSE <группа_операторов>]

END CASE [имя_метки];

```

Блок оператора CASE в PL/SQL может быть помечен меткой для выполнения перехода к нему из других мест кода.

В подпараграфе 6.1.3.1 приведен фрагмент кода в PL/SQL с использованием оператора IF для определения уровня оплаты v_PayLevel в зависимости от суммы оплаты. С использованием CASE пример может быть переписан следующим образом:

```

<<CasePay>>
CASE
    WHEN v_PaySum <= 50
    THEN v_PayLevel := 'Low Pay';
    WHEN v_PaySum > 150 AND v_PaySum <= 300
    THEN v_PayLevel := 'Avg Pay';
    WHEN v_PaySum > 300 AND v_PaySum <= 500
    THEN v_PayLevel := 'Moderate Pay';
    ELSE v_PayLevel := 'High Pay';
END CASE CasePay;

```

Принцип работы оператора CASE в процедурных языках аналогичен принципу работы операции CASE в обычном SQL (см. подп. 3.2.3.5).

6.1.3.3. Операторы циклов

Простой цикл. Из рассматриваемых в настоящем учебнике СУБД простой цикл поддерживается только в СУБД Oracle. Простой цикл реализует повторное выполнение фрагмента кода PL/SQL в следующем виде [1]:

```

LOOP
    <группа_операторов>
END LOOP;

```

Выполнение повторяется до тех пор, пока цикл не будет явным образом завершен с помощью оператора EXIT или EXIT WHEN. Опе-

ратор EXIT немедленно завершает цикл, а оператор EXIT WHEN завершает цикл при истинности заданного условия.

Например, следующий цикл будет выполняться до тех пор, пока переменная v_Counter не превысит значение 15:

```
v_Counter := 1;
LOOP
  v_Counter := v_Counter + 3;
  EXIT WHEN v_Counter > 15;
END LOOP;
```

При организации простого цикла следует внимательно следить за наличием выхода из цикла, чтобы не получить бесконечный цикл.

Цикл WHILE. Для организации цикла с предусловием можно использовать оператор WHILE. Тело цикла WHILE обрабатывается, пока выполняется заданное логическое условие.

Цикл WHILE поддерживается во всех трех рассматриваемых СУБД. В PSQL оператор имеет формат

```
WHILE (логическое_условие) DO <группа_операторов>
```

В T-SQL оператор имеет синтаксис

```
WHILE ([ ] логическое_условие [ ]) <группа_операторов>
```

Логическое условие в T-SQL должно быть заключено в скобки, если представляет собой запрос SELECT.

В PL/SQL синтаксис оператора WHILE:

```
WHILE логическое_условие
LOOP
  <группа_операторов>
END LOOP;
```

Следующий фрагмент кода иллюстрирует использование в PSQL оператора WHILE в предположении, что переменная Result была ранее объявлена и инициализирована как локальная переменная или параметр строкового типа:

```
WHILE (CHAR_LENGTH(Result)<8) DO Result = '0' || Result;
```

Аналогичный код на T-SQL:

```
WHILE (LEN(@Result) < 8) SET @Result = '0' + @Result;
```

на PL/SQL:

```

WHILE LENGTH(Result) < 8
LOOP
    Result := '0' || Result;
END LOOP;

```

В этом фрагменте переданное строковое значение (Result) дополняется нулями слева до получения восьми символов.

Цикл FOR. Различают два варианта реализации цикла FOR в процедурном языке: цикл со счетчиком и цикл неявного курсора. Из рассматриваемых СУБД цикл со счетчиком поддерживает только Oracle, неявный курсор — Firebird и Oracle (см. далее подп. 6.1.4), в MS SQL Server цикл FOR не поддерживается.

Цикл FOR со счетчиком позволяет повторить фрагмент кода заранее определенное число раз и записывается в формате

```

FOR переменная_цикла IN [REVERSE] нижняя_граница .. верхняя_граница
LOOP
    <группа_операторов>
END LOOP;

```

Данный цикл удобно использовать, если известно число итераций, которое должно быть выполнено. Оно вычисляется на основе нижней и верхней границ переменной цикла. Переменная цикла инициализируется значением нижней границы (или, если указана опция REVERSE, значением верхней границы). Затем при каждой итерации переменная цикла инкрементируется (или декрементируется, если указана опция REVERSE). Цикл завершается, когда значение переменной превышает верхнюю границу (или, при наличии опции REVERSE, становится меньше нижней границы).

Ниже приведен пример цикла FOR в PL/SQL, тело которого выполняется 15 раз. Пример не имеет практического смысла, а только демонстрирует использование цикла. В отличие от простого цикла и цикла WHILE, объявлять переменную v_Counter цикла FOR не требуется:

```

FOR v_Counter IN 1..15
LOOP
    Result := v_Counter;
END LOOP;

```

Операторы перехода. В процедурных языках рассматриваемых СУБД существуют операторы, позволяющие:

- 1) выйти из цикла или перейти на начало этого же или другого цикла (LEAVE);
- 2) перейти на конечный оператор END (EXIT);
- 3) временно приостановить выполнение хранимой процедуры для передачи вызвавшей программе полученных данных (SUSPEND);

- в INSERT используется выражение, в котором содержится коллекция таблиц;
- в таблице требуется произвести вставку более чем в 999 столбцов;
- в подзапросе используется последовательность.

Таким образом, возможности запроса INSERT в Oracle значительно шире, чем в Firebird.

В MS SQL Server запрос на вставку данных имеет такой формат [4]:

```
[ <предложение обобщенного табличного выражения> ]
INSERT [ TOP (<выражение>) [ PERCENT ] ]
[ INTO ] { [<сервер>.[<база данных>.<схема>.]
{ <таблица> | <представление> }
| { <функция OPENQUERY> | <функция OPENROWSET> } }
[ WITH (<подсказка таблицы> [, <подсказка таблицы>...]) ]
[ (<список столбцов>) ]
[ <секция_OUTPUT> ]
{ VALUES ( { DEFAULT | NULL | <выражение> } [, { DEFAULT | NULL |
<выражение> } ]...)
[ , ( { DEFAULT | NULL | <выражение> } [, { DEFAULT |
NULL | <выражение> } ]...) ]...
| <запрос_SELECT>
| <оператор_EXECUTE>
| DEFAULT VALUES};
```

где

```
<предложение обобщенного табличного выражения> ::=
WITH <обобщенное табличное выражение>
[ , <обобщенное табличное выражение> ] ... ;
<обобщенное табличное выражение> ::=
<имя_выражения> [ (<столбец1> [, <столбец2>] ...) ]
AS (<запрос_SELECT>)
```

Обобщенное табличное выражение задает временный именованный набор данных, по этому имени к набору данных можно обращаться в запросе INSERT. Набор данных является временным, он существует только в процессе выполнения запроса на вставку данных. Обобщенные табличные выражения были рассмотрены в подп. 3.3.2.2 при изучении подзапросов в секции WITH.

Опция TOP позволяет указать количество либо процент (ключевое слово PERCENT) добавляемых строк, тем самым ограничивая количество вставляемых запросом INSERT данных.

После необязательного ключевого слова INTO указывается, куда именно добавляются строки. Особенностью MS SQL Server является то, что запрос INSERT позволяет добавлять данные не только в таблицы БД, принадлежащей текущему экземпляру сервера, но также в таблицы БД, находящихся в других экземплярах сервера и даже в других

4) досрочно начать новую итерацию цикла (CONTINUE).

Оператор EXIT позволяет из любого места хранимой процедуры, функции или триггера перейти на конечный оператор END, то есть завершить выполнение программы:

```
EXIT [<метка>;
```

Оператор LEAVE осуществляет выход из цикла WHILE независимо от выполнения условия в секции WHILE. Если же в операторе указана метка, то осуществляется переход на начало другого (или того же самого) цикла. Синтаксис оператора

```
LEAVE [<метка>;
```

Если в операторе не указана метка, то оператор просто осуществляет выход из текущего цикла WHILE.

Оператор SUSPEND временно приостанавливает выполнение селективной хранимой процедуры (см. подп. 6.2.3) и передает вызвавшей программе значения выходных параметров.

Синтаксис оператора SUSPEND

```
SUSPEND;
```

Оператор CONTINUE моментально начинает новую итерацию внутреннего цикла оператора WHILE или FOR.

С использованием опционального параметра метки CONTINUE также может начинать новую итерацию для внешних циклов.

Синтаксис оператора CONTINUE

```
CONTINUE [<метка>;
```

6.1.4. Курсоры

При программировании хранимых процедур и триггеров очень часто возникает необходимость выбрать информацию из БД и временно ее сохранить перед выполнением каких-либо других действий. Для таких целей в модулях на процедурных языках допускается использовать запрос на выборку:

```
SELECT ... INTO <список_переменных>;
```

Запрос такого вида можно использовать только в случае, если он возвращает одно-единственное значение (скалярный запрос) или только одну строку из одной или нескольких таблиц БД.

В большинстве случаев запрос к реляционной БД возвращает несколько строк данных, но приложение за одно действие обрабатывает лишь одну строку. При модификации, удалении или добавлении

данных с помощью запросов DML действия также могут выполняться над отдельной строкой таблицы. В этой ситуации на первый план выступает концепция курсора. Под *курсором* понимаются получаемый при выполнении запроса результирующий набор и связанный с ним указатель текущей строки. Курсор — это указатель на приватную область памяти, которая выделяется сервером для обработки SQL-запроса. Обычно курсоры используются для выбора из БД некоторого подмножества хранимой в ней информации. В каждый момент времени прикладной программой может быть обработана одна строка курсора. После позиционирования курсора над этой строкой можно выполнять различные действия.

Курсоры часто применяются в запросах SQL, встроенных в прикладные программы, написанные на языках процедурного типа.

PSQL СУБД Firebird поддерживает два типа курсоров: явные и неявные.

Реализация неявного курсора представлена следующей конструкцией:

```
FOR SELECT ... INTO ... DO ...
```

Она полностью реализует синтаксис цикла и предоставляет возможность последовательной построчной обработки набора данных курсора в цикле FOR (курсорный цикл).

Явный курсор, также называемый изменяемым, или именованным, курсором, представляет собой объект, который разработчик может явно объявлять, открывать и закрывать (в отличие от неявного курсора, не требующего объявления, открытия и закрытия). Для управления явным курсором используются команды DECLARE CURSOR, OPEN, FETCH и CLOSE.

Примечание. Команды OPEN, FETCH и CLOSE, используемые для работы с явным курсором, нельзя применять для работы с неявным курсором.

В MS SQL Server поддерживаются только явные курсоры. При этом следует отметить, что синтаксис реализации явного курсора в T-SQL существенно расширен по сравнению с тем, что предлагает стандарт SQL.

В Oracle, как и в Firebird, поддерживаются и явные, и неявные курсоры. Если запрос SELECT возвращает не более одной строки, то реализуется неявный курсор в виде

```
SELECT ... INTO <список_переменных> ... FROM ... [WHERE ...];
```

Блок PL/SQL, в котором запрос SELECT может возвращать несколько строк, требует использования явного курсора и команд

OPEN, FETCH, CLOSE для работы с ним либо использования специального курсорного цикла FOR для поочередной обработки строк курсора в виде

```
FOR имя_записи IN ... LOOP ... END LOOP;
```

Неявные курсоры создаются PL/SQL автоматически при выполнении SQL-запросов, для которых не объявлен явный курсор.

Независимо от типа курсора работа с ним производится в три этапа:

- 1) открытие (синтаксический анализ, привязка переменных, установка указателя на первую строку, выполнение SQL-запроса);
- 2) выборка (выборка строки, перемещение указателя на следующую строку);
- 3) закрытие (освобождение области памяти, курсор становится недоступным).

Рассмотрим более подробно особенности реализаций двух типов курсоров в различных СУБД.

6.1.4.1. Неявный курсор

Как известно, запрос SELECT возвращает НД, но приложения не всегда могут эффективно работать с результирующим набором. Возникает проблема доступа к каждой строке этой таблицы, которая может быть решена при использовании неявного курсора.

В процедурном языке Firebird цикл для построчной обработки набора данных неявного курсора организуется с помощью конструкции

```
FOR
  SELECT [DISTINCT | ALL]
    { <возвращаемый_элемент1> [, <возвращаемый_элемент2> ] ... }
  FROM базовая_таблица1 [, базовая_таблица2 ] ...
  [WHERE <условие_поиска>]
  [GROUP BY <элемент_группировки1> [, <элемент_группировки2>]... ]
  [HAVING <условие_поиска>]
  [ORDER BY <элемент_сортировки1> [, <элемент_сортировки2>]...]
  INTO :<имя_переменной1> [, :<имя_переменной2> ...]
  [AS CURSOR имя_курсора]
DO <группа_операторов>
```

где

```
<имя_переменной> ::= {имя_локальной_переменной
                        | входной_параметр | выходной_параметр}
```

Как следует из синтаксиса, набор данных курсора определяется запросом SELECT, который может соединять данные из нескольких таблиц, содержать условия поиска, сортировку и т. д.

Для обработки всех строк, возвращаемых запросом SELECT, организуется курсорный цикл. При этом в запросе SELECT используется секция INTO. Она определяет, что значения возвращаемых элементов должны быть присвоены переменным, определенным в теле ХП, триггера или выполняемого блока. В Firebird секция INTO в запросе SELECT должна быть последней, если он имеет другие секции (например, WHERE).

Каждый раз, когда обрабатывается очередная строка в цикле, значения возвращаемых элементов присваиваются переменным, указанным в секции INTO. При этом <группа_операторов> выполняется только единожды для каждой строки, сформированной запросом SELECT. Цикл повторяется, пока не закончатся записи в наборе данных курсора. Например, в теле ХП использование неявного курсора может выглядеть таким образом:

```
DECLARE LAccountCD VARCHAR(6);
DECLARE LFio VARCHAR(20);
DECLARE LPayDate DATE;
DECLARE LPaySum NUMERIC(15,2);
BEGIN
    FOR SELECT A.AccountCD, A.Fio, P.PayDate, P.PaySum
        FROM Abonent A, PaySumma P
        WHERE A.AccountCD = P.AccountCD AND P.PaySum > 70
        INTO :LAccountCD, :LFio, :LPayDate, :LPaySum
    DO
        SUSPEND;
END
```

В данном простом примере в качестве тела цикла используется только оператор SUSPEND. В тот момент, когда выполнение тела цикла доходит до SUSPEND, выполнение приостанавливается для возвращения текущих значений курсора. После возвращения данных (одной строки) прокрутится следующий цикл FOR SELECT до очередного SUSPEND, и так далее, пока курсор не перестанет выводить строки.

В СУБД Oracle построчную обработку набора данных, возвращаемого запросом SELECT, можно организовать с помощью курсорного цикла FOR. Цикл FOR для необъявленного явно курсора реализуется в формате

```
FOR имя_записи IN (<запрос_SELECT>)
LOOP
    <группа_операторов>
END LOOP;
```

Цикл FOR при обработке набора данных курсора помещает текущую строку в запись, к полям которой потом возможно обращение

в теле цикла. Например, цикл FOR курсора на PL/SQL может выглядеть так:

```
DECLARE
    LAccountCD VARCHAR2(6);
BEGIN
    FOR Rec IN (SELECT A.AccountCD
                FROM Abonent A, PaySumma P
                WHERE A.AccountCD = P.AccountCD AND P.PaySum > 70)
    LOOP
        LAccountCD := Rec.AccountCD;
    END LOOP;
END;
```

6.1.4.2. Явный курсор

Работа с явным курсором предполагает выполнение определенного набора действий, стандартных для любой СУБД. В соответствии со стандартом SQL при работе с курсорами выделяются следующие основные действия:

- определение курсора, т. е. его объявление и связывание с ним запроса SELECT;
- открытие курсора, т. е. извлечение строк из БД и помещение их в курсор (в память);
- выборка данных из курсора и обработка выбранных строк (как правило, в цикле);
- закрытие курсора, после чего он становится недоступным;
- освобождение курсора, т. е. удаление курсора как объекта, поскольку его закрытие необязательно освобождает ассоциированную с ним память.

В любой СУБД объявление курсора должно помещаться в начале триггера, ХП или неименованного выполняемого блока, подобно объявлению обычных локальных переменных. Само определение курсора в разных реализациях может иметь некоторые отличия.

Для определения явного курсора и связывания с ним запроса SELECT в Firebird используется оператор

```
DECLARE [VARIABLE] имя_курсора [SCROLL | NO SCROLL] CURSOR
FOR (<запрос_SELECT>);
```

Опция SCROLL после имени курсора определяет *двунаправленный курсор*, который позволяет получать доступ к определенным записям набора данных курсора [6]. Опция NO SCROLL (по умолчанию) объявляет обычный курсор.

Имена курсоров должны быть уникальными в контексте того модуля, где курсоры используются. Иными словами, каждый явный и неявный курсоры (если для последнего объявлено имя с помощью

конструкции AS CURSOR имя_курсора) должны иметь различные имена. Однако допускается совпадение имени курсора с именем любой переменной, используемой в том же самом модуле.

Microsoft SQL Server для определения явного курсора поддерживает синтаксис в двух формах: синтаксис стандарта ISO и синтаксис, использующий набор расширений языка T-SQL [7]. Стандартный синтаксис:

```
DECLARE имя_курсора [ INSENSITIVE ] [ SCROLL ] CURSOR
    FOR <запрос_SELECT>
    [ FOR { READ ONLY | UPDATE [ OF имя_столбца [ ,... ] ] } ] [;]
```

Расширенный синтаксис на T-SQL:

```
DECLARE имя_курсора CURSOR [ LOCAL | GLOBAL ]
    [ FORWARD_ONLY | SCROLL ]
    [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
    [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
    [ TYPE_WARNING ]
    FOR <запрос_SELECT>
    [ FOR UPDATE [ OF имя_столбца [ ,... ] ] ] [;]
```

Расширенная форма позволяет определять курсоры с помощью таких же типов, как типы, используемые в курсорных функциях API БД ODBC и ADO. Подробнее с особенностями использования расширенной формы можно ознакомиться в работах [4, 7], в рамках настоящего учебника будем применять синтаксис ISO для определения курсора в MS SQL Server.

Имя курсора как объекта, в отличие от имен локальных переменных, не должно начинаться с символа @.

Опция INSENSITIVE определяет курсор, который создает временную копию данных для использования. Все запросы к курсору обращаются к указанной временной таблице в БД tempdb, поэтому изменения базовых таблиц не влияют на данные, возвращаемые выборками для данного курсора, а сам курсор не позволяет производить изменения. Если опция не указана, то зафиксированные обновления и удаления, сделанные в базовых таблицах, отображаются в последующих выборках.

Конструкция UPDATE [OF имя_столбца [...]] определяет обновляемые столбцы в курсоре. Если указана опция OF имя_столбца [...], то для изменений будут доступны только перечисленные столбцы. Если конструкция UPDATE используется без списка столбцов, то обновление возможно для всех столбцов.

В СУБД Oracle явный курсор определяется в общем разделе деклараций следующим образом:

```
CURSOR имя_курсора [(<список_параметров>)] [RETURN тип_значения]
```

```
IS <запрос_SELECT>
[ FOR UPDATE [ OF (<список_столбцов>) ] [NOWAIT] ];
```

Необязательный список параметров представляет собой список аргументов, необходимых для работы. Конструкция RETURN определяет тип возвращаемого значения, если таковое предусмотрено. Опция FOR UPDATE блокирует строки БД на то время, когда курсор остается открытым. Опция NOWAIT вызывает завершение программы сразу же после открытия курсора в том случае, если запрос SELECT не может получить исключительный доступ к строкам таблицы.

После того как курсор определен в блоке процедурного языка, необходимо его открыть. Для открытия курсора в любой из рассматриваемых СУБД используется оператор

```
OPEN имя_курсора;
```

Оператор OPEN подготавливает курсор, загружает запрос SELECT и извлекает строки из БД в память, активизируя первую из извлеченных строк.

Для выборки данных из курсора используется оператор FETCH. В Firebird оператор реализован в следующих форматах:

```
FETCH имя_курсора INTO :<имя_переменной1> [, :<имя_переменной2> ...];
FETCH {NEXT | PRIOR | FIRST | LAST | ABSOLUTE <число> | RELATIVE
<число>}
FROM имя_курсора INTO :<имя_переменной1> [, :<имя_переменной2> ...];
```

В MS SQL Server используется оператор FETCH вида

```
FETCH [{ NEXT | PRIOR | FIRST | LAST | ABSOLUTE <число> | RELATIVE
<число> }]
FROM <имя_курсора>
[ INTO @<имя_переменной1> [, @<имя_переменной2>...] ];
```

где

```
<имя_курсора> := { имя_курсора | @имя_переменной }
```

В этом операторе как, впрочем, и в других операторах MS SQL Server для работы с курсорами, можно указывать как сам курсор, так и локальную переменную типа данных CURSOR, ссылающуюся на реальный курсор.

В Oracle выборка данных из курсора выполняется в формате

```
FETCH имя_курсора INTO <имя_переменной1> [, <имя_переменной2> ...];
```

Для закрытия курсора в любой из рассматриваемых СУБД используется оператор

CLOSE имя_курсора;

При этом в различных СУБД по-разному происходит освобождение выделенной для курсора памяти. В некоторых СУБД, например в Firebird, при закрытии курсора освобождение памяти происходит автоматически неявным образом, и сразу она становится доступной для других операций: открытие другого курсора и т. д.

Примечание. В Firebird попытки открыть курсор, который уже открыт, выбрать информацию из уже закрытого курсора или закрыть его вновь будут неудачны. Все курсоры, которые не были закрыты явно с помощью команды CLOSE, будут закрыты автоматически при выходе из текущего PSQL-блока, процедуры или триггера.

В других СУБД необходимо явным образом освободить выделяемую для курсора память. Например, в MS SQL Server для освобождения занимаемой курсором памяти используется оператор

DEALLOCATE имя_курсора;

Рассмотрим примеры работы с явными курсорами в различных СУБД. Например, в Firebird в теле ХП объявление и использование явного курсора могут выглядеть так:

```
DECLARE Aacc VARCHAR(6);
DECLARE Aname VARCHAR(20);
DECLARE c CURSOR FOR
  (SELECT A.AccountCD, A.Fio FROM Abonent A ORDER BY A.AccountCD);
BEGIN
  OPEN c;
  WHILE (1 = 1) DO
  BEGIN
    FETCH c INTO :Aacc, :Aname;
    IF (ROW_COUNT = 0)
    THEN
      LEAVE;
    SUSPEND;
  END
  CLOSE c;
END
```

В данном примере показано, как может быть организована выборка данных с использованием явного курсора в Firebird. Организуется цикл WHILE, внутри которого контекстная переменная ROW_COUNT проверяет, возвращает ли последняя выборка (FETCH) из явного курсора строку данных. Если нет, то происходит выход из цикла WHILE с помощью оператора LEAVE и закрытие курсора.

В MS SQL Server аналогичный код можно реализовать в виде


```

DECLARE @Aacc VARCHAR(6);
DECLARE @Aname VARCHAR(20);
DECLARE c CURSOR FOR
    SELECT AccountCD, Fio FROM Abonent ORDER BY AccountCD
BEGIN
    OPEN c;
    WHILE (1 = 1)
    BEGIN
        FETCH c INTO @Aacc, @Aname;
        IF (@@rowcount = 0)
            BREAK;
    END
    CLOSE c;
    DEALLOCATE c;
END

```

В данном случае для проверки окончания данных курсора используется функция @@rowcount, для выхода из цикла — оператор BREAK, для освобождения памяти курсора после его закрытия — оператор DEALLOCATE.

Для контроля достижения конца курсора в MS SQL Server рекомендуется применять функцию @@FETCH_STATUS [4]. Функция будет возвращать значение 0, пока не будет достигнут конец списка строк, полученных при открытии курсора (пока курсор ссылается на существующую запись набора данных). Когда весь список будет просмотрен, функция вернет значение -1, и цикл будет завершен. В случае если выборка завершилась неудачно вследствие попытки обращения к удаленной или измененной строке, функция вернет значение -2. Поскольку функция @@FETCH_STATUS является глобальной для всех курсоров в соединении, необходимо использовать ее с осторожностью.

Вышеприведенный пример с использованием функции @@FETCH_STATUS в MS SQL Server будет выглядеть так:

```

DECLARE @Aacc VARCHAR(6);
DECLARE @Aname VARCHAR(20);
DECLARE c CURSOR FOR
    SELECT AccountCD, Fio FROM Abonent ORDER BY AccountCD
BEGIN
    OPEN c;
    WHILE @@FETCH_STATUS = 0
    BEGIN
        FETCH c INTO @Aacc, @Aname;
    END
    CLOSE c;
    DEALLOCATE c;
END

```

В СУБД Oracle приведенный пример можно реализовать так:

```

DECLARE
    v_Aacc VARCHAR2(6);
    v_Aname VARCHAR2(20);
    CURSOR c IS
        SELECT A.AccountCD, A.Fio FROM Abonent A ORDER BY A.AccountCD;
BEGIN
    OPEN c;
    LOOP
        FETCH c INTO v_Aacc, v_Aname;
        EXIT WHEN c%NOTFOUND;
    END LOOP;
    CLOSE c;
END;

```

Здесь выборка данных курсора выполняется в простом цикле LOOP...END LOOP. Oracle для определения состояния курсора при работе с ним позволяет использовать специальные курсорные атрибуты: %FOUND (признак удачной выборки), %NOTFOUND (признак отсутствия возвращаемых строк), %ISOPEN (признак открытия курсора) и %ROWCOUNT (количество возвращенных строк). На основе атрибутов формируются специальные логические переменные с именем в формате

имя курсора%атрибут

В данном примере для определения момента окончания цикла используется специальная логическая переменная с%NOTFOUND, которая принимает значение TRUE после того, как оператор FETCH достигает последней доступной строки данных курсора.

Работа с явным курсором в Oracle может быть организована не только с помощью команд OPEN, FETCH и CLOSE, но и с использованием цикла FOR, который выполняет данные операции автоматически в неявном виде. Цикл FOR явного курсора реализуется в формате

```

FOR имя_записи IN имя_курсора
LOOP
    <группа_операторов>
END LOOP;

```

Например, предыдущий код в Oracle с циклом FOR может быть записан в следующем виде:

```

DECLARE
    v_Aacc VARCHAR2(6);
    v_Aname VARCHAR2(20);
    CURSOR c IS
        SELECT A.AccountCD, A.Fio FROM Abonent A ORDER BY A.AccountCD;
BEGIN

```

```

FOR c_rec IN c LOOP
    v_Aacc := c_rec.AccountCD;
    v_Aname := c_rec.Fio;
END LOOP;
END;
```

Система управления БД Firebird и MS SQL Server позволяют использовать в модулях процедурного языка *двунаправленные курсоры*. Двунаправленный курсор, который позволяет получать доступ к определенным записям набора данных курсора, определяет опция SCROLL после имени курсора [6]. Если курсор определен как двунаправленный, то доступны следующие параметры выборки оператора FETCH:

- NEXT возвращает строку в курсоре сразу же за текущей строкой и перемещает указатель текущей строки на возвращенную строку. Если инструкция FETCH NEXT выполняет первую выборку в отношении курсора, то она возвращает первую строку в наборе данных. NEXT является параметром по умолчанию выборки из курсора. Если создается NO SCROLL курсор, то поддерживается только параметр выборки NEXT;
- PRIOR возвращает строку в курсоре, находящуюся непосредственно перед текущей строкой, и перемещает указатель текущей строки на возвращенную строку. Если инструкция FETCH PRIOR выполняет первую выборку из курсора, то не возвращается никакая строка и положение курсора остается перед первой строкой;
- FIRST возвращает первую строку в курсоре и делает ее текущей;
- LAST возвращает последнюю строку в курсоре и делает ее текущей;
- ABSOLUTE <число>. При положительном значении возвращается строка, отстоящая на <число> строк от начала курсора, а возвращенная строка становится новой текущей строкой. Если <число> имеет отрицательное значение, то возвращается строка, отстоящая на <число> строк от конца курсора, а возвращенная строка становится новой текущей строкой. Если <число> равно 0, то не возвращается ни одна строка. Аргумент <число> должен быть целочисленной константой;
- RELATIVE <число>. При положительном значении возвращается строка, отстоящая на <число> строк от текущей строки, а возвращенная строка становится новой текущей строкой. Если <число> имеет отрицательное значение, то возвращается строка, предшествующая на <число> строк текущей строке, а возвращенная строка становится новой текущей строкой. Если <число> равно 0, то возвращается текущая строка. Если при первой выборке из курсора инструкция FETCH RELATIVE указана с аргументом <число>

в отрицательном значении или равном 0, строки не возвращаются. Аргумент <число> должен быть целочисленной константой.

Пример работы явного NO SCROLL курсора для случая использования двунаправленного курсора может быть модифицирован, например в Firebird, таким образом:

```
DECLARE Aacc VARCHAR(6);
DECLARE Aname VARCHAR(20);
DECLARE c SCROLL CURSOR FOR
  (SELECT A.AccountCD, A.Fio FROM Abonent A ORDER BY A.AccountCD);
BEGIN
  OPEN c;
  FETCH FIRST FROM c INTO :Aacc, :Aname;
  SUSPEND;
  FETCH LAST FROM c INTO :Aacc, :Aname;
  SUSPEND;
  CLOSE c;
END
```

В данном примере с помощью двунаправленного курсора сначала выбирается и выводится первая запись набора данных (номер лицевого счета и ФИО абонента с наименьшим номером лицевого счета), а затем последняя (номер лицевого счета и ФИО абонента с наибольшим номером лицевого счета).

Следует отметить еще одну важную особенность работы с курсорами. Они позволяют обеспечить обновление или удаление строки, на которой в данный момент установлен курсор, с помощью конструкции WHERE CURRENT OF в запросе UPDATE или DELETE. В этом случае <группа_операторов> при обновлении строки, на которой установлен курсор, должна выглядеть так:

```
UPDATE базовая_таблица SET ... WHERE CURRENT OF имя_курсора;
```

При удалении строки, на которой установлен курсор, <группа_операторов> должна выглядеть таким образом:

```
DELETE FROM базовая_таблица WHERE CURRENT OF имя_курсора;
```

За одну операцию обновления могут быть изменены несколько столбцов текущей строки курсора, но все они должны принадлежать одной таблице. Если выполняется удаление, то будет удалена строка, установленная текущей в курсоре.

В заключение описания курсоров в процедурных языках следует отметить, что в некоторых случаях, конечно, применение курсора неизбежно, однако по возможности этого следует избегать и работать со стандартными запросами обработки данных (SELECT, UPDATE, INSERT, DELETE). Помимо того, что курсоры не позволяют проводить операции изменения над всем объемом данных, ско-

рость выполнения операций обработки данных посредством курсора заметно ниже, чем у стандартных средств SQL.

6.1.5. Последовательности

В процедурных языках существует очень важный с точки зрения практического использования БД механизм порождения уникальных значений — последовательности.

Последовательностью называется специальный объект БД для получения целочисленных значений, следующих с определенным шагом. Последовательности часто применяются для уникальной идентификации строк таблиц. Ценность автоматической генерации уникальных идентификаторов становится очевидной, если рассмотреть, что может произойти, когда информацию в БД одновременно вводят десятки или сотни пользователей. Для каждой новой вводимой строки необходимо гарантировать, что введенный первичный ключ отличается от всех остальных чисел, присваиваемых другими пользователями. Последовательность решает эту задачу, мгновенно раздавая разные целые числа всем затребовавшим пользователям.

Следует отметить, что в Firebird как аналог последовательности часто используется термин *генератор*.

Рассмотрим принципы работы с последовательностями значений в различных СУБД. В БД Firebird для создания или пересоздания генератора используется запрос

```
{CREATE | RECREATE } { SEQUENCE | GENERATOR } имя_последовательности  
[START WITH целое_число] [INCREMENT [BY] шаг];
```

Необязательная опция INCREMENT BY задает целое число (шаг), на которое увеличивается (или уменьшается, если оно отрицательное) значение объекта последовательности при каждом вызове функции NEXT VALUE FOR. Если значение приращения отрицательно, то объект последовательности убывает, в противном случае — возрастает. Шаг не может быть равен 0 (по умолчанию равен 1).

По стандарту SQL для обозначения генератора последовательности используется синтаксический термин SEQUENCE, в то время как GENERATOR — это синтаксический термин InterBase. Для соответствия SQL-стандарту для работы с генераторами последовательностей рекомендуется использовать именно термин SEQUENCE.

Каждая последовательность в Firebird, как и в других СУБД, имеет уникальное имя и текущее значение. Опция START WITH используется для установки определенного (текущего) значения генератора целое_число. При этом целое_число — это значение типа BIGINT, которое должно лежать в пределах от -2^{63} до $2^{63} - 1$. Если при создании генератора секция START WITH не задана, то по умолчанию генератор создается с текущим значением 0. В таком случае

в дальнейшем для установки текущего значения генератора можно использовать команду

```
SET { SEQUENCE | GENERATOR } имя_последовательности TO целое_число;
```

Однако для соответствия стандарту SQL при установке последовательности (генератора) в заданное значение, в том числе и то, которое было установлено при создании последовательности (или при предыдущей установке значения), и (или) изменении шага предпочтительнее использовать запрос

```
ALTER { SEQUENCE | GENERATOR } имя_последовательности  
[RESTART [WITH целое_число]] [INCREMENT [BY] шаг];
```

Для создания/изменения определения генератора в Firebird также может использоваться запрос

```
CREATE OR ALTER { SEQUENCE | GENERATOR } имя_последовательности  
[{ RESTART | START WITH целое_число }] [INCREMENT [BY] шаг];
```

В этом случае должна присутствовать хотя бы одна из необязательных секций.

В СУБД MS SQL Server последовательность создается с помощью запроса

```
CREATE SEQUENCE [имя_схемы.] имя_последовательности  
[ AS тип_данных ]  
[ START WITH целое_число ]  
[ INCREMENT BY шаг ]  
[ { MINVALUE [ целое_число ] | NO MINVALUE } ]  
[ { MAXVALUE [целое_число ] | NO MAXVALUE } ]  
[ { CYCLE | NO CYCLE } ]  
[ { CACHE [целое_число ] | NO CACHE } ] [ ; ]
```

Последовательность может быть определена с любым целочисленным типом (по умолчанию — BIGINT).

Опции MAXVALUE и MINVALUE задают соответственно минимальное и максимальное граничные значения последовательности. Бессмысленно задавать MINVALUE при положительных, а MAXVALUE — при отрицательных значениях INCREMENT BY.

Опции цикличности CYCLE/NO и CYCLE указывают, перезапускается ли объект последовательности с минимального (или максимального для убывающих последовательностей) значения или вызывает исключение, когда достигнуто максимальное (или минимальное) значение (NO CYCLE — по умолчанию).

Опция CACHE задает размер кэша, повышая тем самым производительность для приложений, использующих объекты последова-

тельностью, сводя к минимуму число операций дискового ввода-вывода, которые требуются для создания порядковых номеров.

Последовательности в MS SQL Server создаются в системной таблице INFORMATION_SCHEMA.SEQUENCES. При создании новой последовательности в таблицу добавляется строка структуры: имя последовательности, схема, каталог, тип данных и все свойства последовательности.

Для изменения параметров последовательности после ее создания в MS SQL Server используется запрос вида

```
ALTER SEQUENCE [имя_схемы.] имя_последовательности
[ RESTART [ WITH целое_число ] ]
[ INCREMENT BY шаг ]
[ { MINVALUE целое_число | NO MINVALUE } ]
[ { MAXVALUE целое_число | NO MAXVALUE } ]
[ { CYCLE | NO CYCLE } ]
[ { CACHE [целое_число ] | NO CACHE } ] [ ; ]
```

Тип данных последовательности нельзя изменить с помощью запроса ALTER SEQUENCE.

В Oracle синтаксис запроса создания последовательностей очень схож с синтаксисом MS SQL Server:

```
CREATE SEQUENCE имя_последовательности
[ START WITH целое_число ]
[ INCREMENT BY шаг ]
[ { MAXVALUE целое_число | NOMAXVALUE } ]
[ { MINVALUE целое_число | NOMINVALUE } ]
[ { CYCLE | NOCYCLE } ]
[ { ORDER | NOORDER } ]
[ { CACHE целое_число | NOCACHE } ];
```

Опция ORDER задает, что целые числа должны распределяться в порядке поступления запросов на них: в ответ на первый запрос выдается ближайшее свободное число, в ответ на второй — следующее и т. д. (по умолчанию задано NOORDER).

Опции CACHE/NOCACHE задают, может ли Oracle заблаговременно сгенерировать ряд уникальных значений до того момента, как они потребуются, и записать их в памяти для быстрой выдачи в ответ на запрос. Опция NOCACHE (по умолчанию) означает, что предварительно группа чисел генерироваться не будет.

Параметры последовательности можно изменить с помощью запроса ALTER SEQUENCE, упрощенный синтаксис которого выглядит так:

```
ALTER SEQUENCE имя_последовательности
[ INCREMENT BY шаг ]
[ { MAXVALUE целое_число | NOMAXVALUE } ]
```

```
[ { MINVALUE целое_число | NOMINVALUE } ]  
[ { CYCLE | NOCYCLE } ]  
[ { ORDER | NOORDER } ]  
[ { CACHE целое_число | NOCACHE } ];
```

Любые изменения последовательности применяются к значениям, генерируемым после модификации. Особо следует отметить ограничения, накладываемые на изменения последовательностей в Oracle: значение START WITH нельзя изменять, и любые изменения не могут сделать недействительными существующие значения последовательности. Например, нельзя изменить последовательность, понизив MAXVALUE до значения, меньшего уже сгенерированного числа последовательности.

Создание генератора последовательности и установка начального значения могут быть произведены при выполнении описанных выше запросов в SQL-редакторе графической утилиты для работы с БД (в IBExpert, dbForge Studio и т. д.) или при использовании этих запросов в тексте SQL-сценария.

Например, последовательность с именем Executor_ID, которая может использоваться для автоматической генерации очередного значения суррогатного первичного ключа таблицы Executor учебной БД, можно определить в любой из рассматриваемых СУБД с помощью запроса

```
CREATE SEQUENCE Executor_ID;
```

Установка начального значения равным 6 для созданной последовательности Executor_ID в Firebird и MS SQL Server может быть выполнена так:

```
ALTER SEQUENCE Executor_ID RESTART WITH 6;
```

В Oracle такой запрос выполнить нельзя, и начальное значение для последовательности, если оно отличается от 0, должно устанавливаться сразу при создании:

```
CREATE SEQUENCE Executor_ID START WITH 6;
```

После создания и установки начального значения последовательности при работе с БД требуется получать очередное (следующее) значение последовательности. В Firebird получение очередного значения генератора производится вызовом функции GEN_ID или функции NEXT VALUE FOR. Функция GEN_ID имеет формат

```
GEN_ID (имя_последовательности, шаг)
```

где шаг представляет собой целое значение (если оно равно нулю, то будет получено текущее значение генератора).

В MS SQL Server и Oracle функция GEN_ID не поддерживается. Функция NEXT VALUE FOR имеет формат

NEXT VALUE FOR имя_последовательности

Значение шага при использовании данной функции всегда равно значению, заданному в опции INCREMENT [BY] при создании последовательности. Если при создании последовательности не указывалась эта опция, то значение шага при использовании функции NEXT VALUE FOR равно 1.

В MS SQL Server для получения очередного значения последовательности также используется функция NEXT VALUE FOR.

В Oracle для обращения к значениям последовательности используются два псевдостолбца: NEXTVAL и CURRVAL. На первый взгляд псевдостолбец Oracle кажется столбцом из таблицы БД, но на самом деле это команда, возвращающая значение. NEXTVAL — это псевдостолбец, который в действительности генерирует очередное (следующее) значение последовательности, используется в виде

<имя_последовательности>.NEXTVAL

После того как очередное значение создано, оно записывается в псевдостолбце CURRVAL (текущее значение). Данный псевдостолбец очень полезен, поскольку можно обратиться к нему и посмотреть, какое значение сгенерировала БД, а затем, например, заполнить столбец внешнего ключа дочерней таблицы с помощью только что сгенерированного значения.

Для того чтобы обратиться к последнему сгенерированному значению, используется следующий синтаксис:

<имя_последовательности>.CURRVAL

Функции вызова генератора в любой СУБД могут применяться:

- 1) для получения очередного значения последовательности и присвоения его локальной переменной в теле триггера или хранимой процедуры;
- 2) в списке возвращаемых элементов запроса SELECT;
- 3) при добавлении и изменении строк (INSERT- и UPDATE-запросы DML).

Например, для созданного генератора Executor_ID получение очередного значения и присвоение его локальной переменной Executor_Number (в теле триггера или хранимой процедуры) в Firebird имеют вид

```
Executor_Number = GEN_ID(Executor_ID, 1);
```

Функции получения значения последовательности могут использоваться в списке возвращаемых элементов запроса SELECT.

В Firebird в таком случае в секции FROM запроса SELECT указывается системная таблица RDB\$DATABASE. Например, для получения текущего значения генератора Executor_ID в списке возвращаемых элементов запроса SELECT можно использовать запрос

```
SELECT GEN_ID (Executor_ID,0) FROM RDB$DATABASE;
```

Результат выполнения этого запроса представлен на рис. 6.2.

GEN_ID
6

Рис. 6.2. Результат получения текущего значения генератора

В Oracle в таком случае в секции FROM запроса SELECT указывается системная таблица DUAL:

```
SELECT Executor_ID.CURRVAL FROM DUAL;
```

В MS SQL Server получить текущее значение можно из системной таблицы SYS.Sequences:

```
SELECT CURRENT_VALUE
FROM SYS.Sequences
WHERE object_id = object_id('dbo.Executor_ID');
```

Для получения следующего значения созданного генератора Executor_ID с помощью функции NEXT VALUE FOR в Firebird нужно выполнить запрос

```
SELECT NEXT VALUE FOR Executor_ID FROM RDB$DATABASE;
```

В результате выполнения этого запроса будет выдано значение, равное 7.

В MS SQL Server запрос будет выглядеть так:

```
SELECT NEXT VALUE FOR Executor_ID;
```

а в Oracle —

```
SELECT Executor_ID.NEXTVAL FROM DUAL;
```

Примером использования генератора последовательности в DML-запросе на вставку строки в Firebird и MS SQL Server может служить запрос

```
INSERT INTO Executor
VALUES (NEXT VALUE FOR Executor_ID, 'Иванов А. А.');
```

В Oracle запрос будет следующим:

```
INSERT INTO Executor VALUES (Executor_ID.NEXTVAL, 'Иванов А. А.');
```

Следует отметить, что в Firebird вызов функции вида

```
GEN_ID (имя_последовательности, 0)
```

никогда не должен использоваться в запросах DML на добавление или изменение данных.

Созданную и неиспользуемую последовательность можно удалить путем выполнения запроса DROP SEQUENCE. В Firebird запрос имеет формат

```
DROP { SEQUENCE | GENERATOR } имя_последовательности;
```

В MS SQL Server и Oracle:

```
DROP SEQUENCE имя_последовательности;
```

Во всех рассматриваемых СУБД генераторы работают вне контекста транзакций. Это означает, что однажды сгенерированное число установлено и не может быть изменено отменой транзакции [5].

В подпараграфе 5.4.2 обсуждалась проблема формирования значений первичного ключа при добавлении в таблицу новых строк. Приведенные там примеры решений можно использовать только в частном случае, а именно при добавлении только одной строки, причем в однопользовательском режиме.

Приведем возможный вариант решения задачи с использованием генератора последовательности. Пусть требуется удалить все погашенные ремонтные заявки, поданные абонентами, проживающими не только на улице с кодом 7, но и на улице с кодом 4. Если у абонента нет зарегистрированных заявок, то следует добавить в таблицу Request информацию о новой заявке.

Запрос MERGE на слияние данных в Firebird:

```
MERGE INTO Request R
USING (SELECT * FROM Abonent WHERE StreetCD = 7 OR StreetCD = 4) A
ON R.AccountCD = A.AccountCD
WHEN MATCHED AND R.Executed THEN DELETE
WHEN NOT MATCHED THEN INSERT (RequestCD, AccountCD, FailureCD,
                               IncomingDate, Executed)
VALUES (NEXT VALUE FOR Gen_Request, A.AccountCD, 12,
        CURRENT_TIMESTAMP, False);
```

В результате выполнения запроса будут удалены 6 заявок и добавлены 2 неназначенные, невыполненные и непогашенные заявки с кодами 24 и 25 соответственно.

В учебной БД определено семь генераторов последовательностей соответственно для каждой из таблиц, в которых первичный ключ является суррогатным. Примеры использования этих генераторов рассмотрены далее при изучении триггеров в подпараграфе 6.6.

6.1.6. Обработка исключительных ситуаций

Исключительные ситуации (Exception) — это сообщения, которые генерируются при появлении ошибок в теле программного модуля.

Исключительные ситуации и их обработчики — это метод, с помощью которого программа реагирует на ошибки выполнения и устраняет их. В случае возникновения ошибки инициируется (raised) исключительная ситуация. При этом управление программой передается обработчику исключительной ситуации, который является отдельным фрагментом программы. Необходимо отметить, что раздел обработки ошибок отделен от остальной части программы, что упрощает логическую структуру программы и обеспечивает фиксацию всех ошибок. В этом случае:

- 1) логика программы нагляднее и легче для понимания;
- 2) программа распознает и обрабатывает любую ошибку независимо от того, какой оператор является причиной ошибки.

Однако следует учесть, что выполнение программы не будет продолжаться с оператора, вызвавшего ошибку. Вместо этого выполнение будет продолжаться в обработчике исключительных ситуаций, а затем во внешнем блоке.

При использовании механизма исключительных ситуаций написанные на процедурном языке программы становятся гораздо надежнее, так как во время их выполнения предоставляется возможность обработки как запланированных, так и незапланированных ошибок. Не следует допускать возникновения в программах исключительных ситуаций, которые не обрабатываются.

Существуют два основных типа исключений: *предопределенные* (системные, встроенные) и *пользовательские*. *Предопределенные исключения* и ассоциированные с ними тексты сообщений связаны с типичными ошибками, возникающими при работе с БД. К таким ошибкам относятся, например, попытка удалить объект, который находится в использовании, попытка вставить некорректные данные в столбец, на который наложено ограничение, деление на ноль и т. д. Предварительно определенные исключения существуют во всех СУБД. Явного объявления и обработки с помощью процедурного языка такие исключения не требуют.

Кроме того, во многих СУБД существует возможность создания *пользовательских исключений*. Исключения, заданные пользователем, — это ошибки, которые предусмотрел пользователь, и они не обязательно связаны с ошибками СУБД. Такие исключения в не-

которых случаях рассматриваются разработчиками как удобный способ передачи управления другому разделу блока процедурного языка. Именованные пользовательские исключения поддерживаются в Firebird и Oracle Database, в MS SQL Server существует возможность генерации пользовательских сообщений об ошибках. Таким образом, создавать пользовательские исключения можно как в клиентском приложении, так и на стороне сервера средствами SQL.

Система управления БД Oracle в дополнение к основным типам исключений поддерживает промежуточный тип — *непредопределенные исключения*, возбуждаемые сервером. Такие исключения, как и предопределенные, вызываются стандартными ошибками сервера и не требуют явного вызова, но при этом, как пользовательские, требуют явного определения.

Рассмотрим особенности работы с исключениями в процедурных языках различных СУБД. Различают объявление (определение), инициирование и обработку исключительной ситуации.

В Firebird пользовательские исключения могут быть *простыми* и *параметризованными* (доступны, начиная с версии Firebird 3.0). Для определения нового пользовательского исключения в Firebird употребляется запрос CREATE EXCEPTION, имеющий формат

```
CREATE EXCEPTION имя_исключения <сообщение>;
```

где <сообщение> — строка текста длиной до 1021 символов, заключенная в апострофы.

Например, чтобы создать простое пользовательское исключение с именем ExcErr и сообщением 'Ошибка', необходим запрос

```
CREATE EXCEPTION ExcErr 'Ошибка';
```

В случае создания параметризованного исключения <сообщение> внутри текста содержит набор параметров в виде @N, где N (параметр) представляет собой число, начиная с 1.

Создать параметризованную исключительную ситуацию можно, например, с помощью запроса

```
CREATE EXCEPTION ExcErrPar 'Неверное значение @1 для столбца @2';
```

Для изменения текста сообщения служит запрос ALTER EXCEPTION:

```
ALTER EXCEPTION ExcErr 'Ошибка преобразования';
```

Если требуется заново создать исключение со старым именем, то используется запрос RECREATE EXCEPTION. Если исключение не существует перед использованием этого запроса, то его исполь-

зование эквивалентно выполнению запроса CREATE EXCEPTION. Уже существующее исключение удаляется запросом RECREATE EXCEPTION (если оно не используется другими объектами) и создает полностью новый объект, например в случае

```
RECREATE EXCEPTION ExcErr 'Ошибка преобразования данных';
```

Также может использоваться запрос CREATE OR ALTER EXCEPTION, который создает исключение, если оно не существует, или изменяет определение существующего исключения.

Для удаления исключительной ситуации используется запрос DROP EXCEPTION, например:

```
DROP EXCEPTION ExcErr;
```

Следует отметить, что создать, пересоздать, изменить или удалить исключительную ситуацию можно как при выполнении соответствующего запроса в SQL-редакторе IBExpert, так и в тексте SQL-сценария, а также с помощью конструктора, вызываемого по Ctrl + N на закладке **Исключения** инспектора объектов БД.

Обработка исключений выполняется в коде процедурных модулей с помощью специальных команд и операторов. Код может перехватывать ошибки при их появлении и затем их обрабатывать: исправлять, реализовывать обход ошибки, выдавать пользователю необходимое сообщение.

В Firebird, чтобы активизировать в программном модуле генерацию сообщения об исключительной ситуации и тем самым прервать выполнение, например триггера или хранимой процедуры, необходимо использовать команду

```
EXCEPTION имя_исключения [USING (<список параметров>)];
```

При этом действия, которые были произведены в триггере или хранимой процедуре над данными в текущей транзакции, отменяются и выполнение ХП или триггера прекращается.

Опция USING используется в случае вызова параметризованного исключения с целью передачи значений входных параметров [6, 15, 16]. Например, созданное выше исключение ExcErrPar может быть вызвано внутри ХП с входным параметром ThisMonth:

```
IF (ThisMonth > 12) THEN  
  EXCEPTION ExcErrPar USING (ThisMonth, 'PayMonth');  
ELSE  
  PayMonth = ThisMonth;  
END
```

Если при вызове параметры не будут заданы, текст сообщения будет выведен в первоначальном виде. Если в качестве значения па-

раметра будет передано значение NULL, то будет выдано сообщение вида '... null...'. Если при вызове передано больше параметров, чем задано при создании исключения, то они игнорируются.

Для простого исключения текст сообщения можно заранее и не создавать, а генерировать исключительную ситуацию с сообщением, задаваемым во время выполнения. Активизация существующего исключения по формату

```
EXCEPTION имя_исключения <сообщение>;
```

заменяет текст исходного сообщения альтернативным текстом <сообщение>.

Для обнаружения и обработки системных и пользовательских исключений в хранимой процедуре или триггере Firebird может также использоваться оператор WHEN:

```
WHEN {SQLCODE код_ошибки_SQL | GDSCODE код_ошибки_Firebird | ANY}  
DO {EXCEPTION имя_исключения | блок_операторов};
```

где код_ошибки_SQL и код_ошибки_Firebird — коды ошибок [5,15].

Оператор WHEN перехватывает ошибки и исключения в текущем блоке операторов, а также во вложенных блоках, если эти ошибки и исключения не были в них обработаны. Областью видимости оператора WHEN ... DO являются все изменения, произведенные до оператора вызвавшего ошибку [15].

Например, следующий блок кода Firebird завершается группой обработчиков исключений:

```
BEGIN  
...  
WHEN SQLCODE -802 DO  
    EXCEPTION ExcErr;  
WHEN SQLCODE -803 DO  
    EXCEPTION ExcErrPar USING (ThisMonth, 'PayMonth');  
WHEN ANY DO  
    EXECUTE PROCEDURE P_ANY_EXCEPTION (SQLCODE);  
END
```

Первые два оператора WHEN обрабатывают ошибки SQLCODE в виде пользовательских исключений ExcErr и ExcErrPar. Если не появилось ни одного из предусмотренных исключений, а было некоторое другое, то его перехватит оператор WHEN ANY. Его обработчик вызывает хранимую процедуру P_ANY_EXCEPTION, передавая в качестве входного параметра код SQLCODE из контекста PSQL-блока. В процедуре может, например, выполняться вывод записи протокола ошибок.

В триггере, ХП или ХФ может присутствовать произвольное количество операторов WHEN. Каждый должен размещаться в самом

конце программного блока перед оператором END. Число операторов WHEN, следующих один за другим и обрабатывающих различные группы ошибок или пользовательских исключений [36], может быть произвольным.

При создании учебной БД Firebird определены три пользовательских исключения: Ins_Restrict, Del_Restrict и Upd_Restrict. Примеры использования этих исключений, а также примеры определения и использования других исключений подробно рассмотрены далее при изучении триггеров в параграфе 6.6.

В Oracle исключения, заданные пользователем, не создаются как отдельный объект БД, а объявляются в разделе деклараций блока процедурного языка с помощью конструкции

```
имя_исключения EXCEPTION;
```

Вызвать исключение, определенное пользователем, можно с помощью оператора RAISE в формате

```
RAISE имя_исключения;
```

Непредопределенные исключения, возбуждаемые сервером, требуют объявления и связывания со стандартной ошибкой Oracle в разделе деклараций. Синтаксис объявления непредопределенного исключения такой же, как и пользовательского, а для связывания ошибки Oracle с именем исключения используется директива компилятора PRAGMA EXCEPTION_INIT. Синтаксис связывания исключения и кода ошибки Oracle следующий:

```
PRAGMA EXCEPTION_INIT (имя_исключения, номер_ошибки);
```

где имя_исключения — объявленное исключение, номер_ошибки — стандартный код ошибки сервера.

Например, объявление и связывание непредопределенного исключения для ошибки ORA-02292 (нарушение правила целостности) внутри раздела деклараций может выглядеть так:

```
RuleErr EXCEPTION;  
PRAGMA EXCEPTION_INIT(RuleErr, -2292);
```

Стандартные исключения иницируются автоматически при возникновении соответствующей ошибки Oracle. При установлении исключительной ситуации управление программой сразу же передается специальному разделу обработки исключений — EXCEPTION. Данный раздел помещается внутри тела процедурного блока и содержит код, предназначенный для поддержки исключений, в виде

```
EXCEPTION  
WHEN имя_исключения1 [OR имя_исключения2 ...] THEN
```



```

        <группа_операторов1>;
[WHEN имя_исключения3 [OR имя_исключения4 ...] THEN
    <группа_операторов2>;]
[WHEN OTHERS THEN
    <группа_операторов3>;]

```

Каждый обработчик исключения состоит из ключевого слова WHEN, задающего исключение, за которым следует последовательность команд, выполняемых в случае, если исключение возбуждено. Ключевое слово OTHERS обозначает обработку всех исключений, не перечисленных явно в WHEN. Таким образом, задается обработчик непредусмотренных исключений. Он используется в основном тогда, когда нельзя полностью понять суть исключений, которые могут возникнуть в блоке PL/SQL.

Например, если необходимо обработать все непредвиденные ошибки, код в Oracle может выглядеть так:

```

BEGIN
    ...
EXCEPTION
    WHEN OTHERS THEN
        EXECUTE PROCEDURE P_ANY_EXCEPTION (SQLCODE);
END

```

Следует учесть, что если возникает исключение, то перед выходом из блока PL/SQL выполняется только один обработчик исключений. После передачи управления обработчику невозможно вернуться в выполняемый раздел блока.

В MS SQL Server возможность перехвата и обработки ошибок реализуется с помощью конструкции TRY/CATCH в формате

```

BEGIN TRY
    <группа_операторов1>
END TRY
BEGIN CATCH
    [ <группа_операторов2>]
END CATCH;

```

В блок TRY помещается код, в котором возможно возникновение исключительной ситуации, а в блоке CATCH — действия в случае ее возникновения. В процессе обработки ошибок можно использовать следующие системные функции для получения подробных сведений об ошибке:

1) ERROR_NUMBER() — возвращает номер ошибки (целое число), если вызвана из блока CATCH, иначе возвращает NULL. Номер ошибки также можно получить вызовом системной переменной @@ERROR;

- 2) `ERROR_SEVERITY()` — возвращает степень серьезности ошибки (целое число), если вызвана из блока `CATCH`, иначе возвращает `NULL`;
- 3) `ERROR_STATE()` — возвращает код состояния ошибки (целое число), если вызвана из блока `CATCH`, иначе возвращает `NULL`;
- 4) `ERROR_PROCEDURE()` — имя хранимой процедуры или триггера, где произошла ошибка. Если ошибка произошла вне процедуры или триггера, то возвращает `NULL`;
- 5) `ERROR_LINE()` — номер строки в триггере или процедуре, где была обнаружена ошибка;
- 6) `ERROR_MESSAGE()` — возвращает полный текст сообщения об ошибке (до 4000 символов), если вызвана из блока `CATCH`, иначе возвращает `NULL`.

Например, перехват и вывод информации об ошибке в коде T-SQL может выглядеть так:

```
BEGIN TRY
    SELECT 1 / 0;
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS ErrorNumber,
           ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
```

Результат выполнения блока представлен на рис. 6.3.

ErrorNumber	ErrorMessage
8134	Обнаружена ошибка: деление на ноль

Рис. 6.3. Результат работы TRY/CATCH

Конструкция `TRY/CATCH` не может быть использована в функциях, определенных пользователем.

Для генерации исключения и передачи управления блоку `CATCH` конструкции `TRY/CATCH` в MS SQL Server может использоваться оператор `THROW` в формате

```
THROW [ {<номер ошибки>|имя_переменной},
        {<сообщение>|имя_переменной},
        {<состояние>|имя_переменной} ];
```

где `<номер ошибки>` — это номер ошибки, определенной пользователем (значение может находиться в диапазоне от 50 000 до 2 147 483 647, ошибки с номерами менее 50 000 — это системные ошибки); `<сообщение>` — строка длиной до 2048 символов; `<состояние>` — число в диапазоне от 0 до 255.

Оператор THROW без параметров генерирует пустое исключение.

Например, можно сгенерировать пользовательское сообщение об ошибке внутри блока с входным параметром ThisMonth:

```
BEGIN TRY
    IF (@ThisMonth > 12)
        THROW 50001, 'Недопустимое значение параметра Месяц', 1;
    ELSE
        SET @PayMonth = @ThisMonth;
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS ErrorNumber,
           ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
```

Результат выполнения блока представлен на рис. 6.4.

ErrorNumber	ErrorMessage
50001	Недопустимое значение параметра Месяц

Рис. 6.4. Результат работы THROW

6.2. Хранимые процедуры

Хранимой процедурой (stored procedure) называется скомпилированная подпрограмма произвольной длины на процедурном языке СУБД, которая хранится в БД как часть метаданных. Хранимая процедура — это именованный блок процедурного языка, содержащий определенный набор инструкций. Он обычно пишется в виде последовательности SQL-команд, выполняющих некоторую задачу [1]. Хранимые процедуры похожи на обыкновенные процедуры языков высокого уровня, они могут иметь входные и выходные параметры, использовать локальные переменные, внутри может быть объявлена другая процедура или функция (параграф 6.3). В них есть возможность производить числовые вычисления и операции над символьными данными, результаты которых могут присваиваться переменным и параметрам. Кроме того, в ХП можно выполнять стандартные операции с БД. Особенно важно то, что в ХП возможны циклы и ветвления. Хранимая процедура может быть вызвана с клиента, из другой процедуры, функции, выполнимого блока или триггера [15].

Использование ХП имеет свои преимущества [5, 17]:

1) *производительность*. При создании текст процедуры оптимизируется и хранится в БД в скомпилированном виде. В таком виде процедура выполняется гораздо быстрее, чем в случае динамическо-

го компилирования каждой составляющей ее команды. Выполняются ХП сервером, а не клиентом, что позволяет сократить сетевой трафик;

2) *модульное проектирование*. Использование хранимых процедур часто позволяет значительно сократить объем кода клиентского приложения. Приложения, получающие доступ к одной и той же БД, могут совместно использовать одни и те же процедуры, и не нужно снова программировать одни и те же действия, благодаря чему уменьшается риск программных ошибок в клиентских приложениях и снижается время разработки новых программных продуктов;

3) *локализация изменений*. Если процедура модифицируется, то все внесенные изменения автоматически отражаются во всех приложениях, использующих процедуру, обеспечивая их согласованность;

4) *защита*. В большинстве СУБД ХП считаются защищаемыми объектами, и им назначаются отдельные привилегии. Вместо того чтобы предоставить пользователю доступ к таблице БД напрямую, можно дать права ХП. Пользователь, вызывающий ХП, должен иметь право на ее выполнение;

5) *простота доступа*. В больших БД набор ХП может стать для прикладных программ основным средством доступа к БД. Часто вызов стандартной процедуры более понятен, чем выполнение последовательности SQL-команд (запросов);

6) *реализация деловой логики*. Возможности условной обработки, предоставляемые ХП, часто используются для реализации бизнес-логики в БД. **Бизнес-логика** — это совокупность правил, принципов, зависимостей поведения объектов предметной области (области человеческой деятельности, которую система поддерживает). Другими словами, бизнес-логика (деловая логика предметной области) — это реализация правил и ограничений автоматизируемых операций, реализация предметной области в информационной системе. К ней относятся, например, формулы расчета ежемесячных плат за жилищно-коммунальные услуги (в информационных системах ЖКХ), автоматизированная отправка сообщений электронной почты руководителю проекта по окончании выполнения частей задания всеми подчиненными (в системах управления проектами), отказ от отеля при отмене рейса авиакомпанией (в туристическом бизнесе) и т. д.

В ХП можно реализовывать сложные алгоритмы обработки данных внутри базы, причем во многих случаях использование ХП позволяет четко отделять алгоритмы логики программы от алгоритмов обработки данных.

6.2.1. Определение, модификация и удаление процедур

Реализация хранимых процедур варьируется от одной СУБД к другой. Различают создание ХП и их вызов. В Firebird хранимая процедура может быть создана в IBExpert после подключения к БД:

- 1) с помощью конструктора, вызываемого по Ctrl + N на закладке **Процедуры** инспектора объектов БД;
- 2) из пункта **Новая процедура** главного меню **База данных**;
- 3) выполнением запроса в SQL-редакторе или в скрипте.

В Firebird хранимые процедуры создаются запросом CREATE PROCEDURE, имеющим синтаксис

```
CREATE PROCEDURE имя_процедуры
  [(<список_входных_параметров>)]
  [RETURNS (<список_выходных_параметров>)]
  AS <тело_процедуры>
```

где

```
<список_входных_параметров> ::= входной_параметр1 <тип_данных>
                                [,входной_параметр2 <тип_данных> ... [= <значение>]],
<список_выходных_параметров> ::= выходной_параметр1 <тип_данных>
                                [,выходной_параметр2 <тип_данных> ...]
```

Входные параметры хранимой процедуры указываются в круглых скобках после имени создаваемой процедуры. Причем в качестве входных параметров могут использоваться выражения. При объявлении входных параметров допускается задавать значения по умолчанию, например:

```
CREATE PROCEDURE p1(x INTEGER = 123)
  RETURNS (y INTEGER) AS ...
```

Примечание. Параметры со значениями по умолчанию должны быть указаны последними в списке входных параметров, т. е. нельзя объявить параметр, не имеющий значение по умолчанию, после параметров, объявленных со значением по умолчанию.

Подстановка значения по умолчанию происходит во время выполнения процедуры.

Секция RETURNS предназначена для определения выходных параметров, используемых для возврата данных из ХП.

Тело процедуры реализуется в виде

```
<тело_процедуры> ::=
  [<список_объявления>]
  <блок_операторов>
```

где

```
<список_объявления> ::=
  [ DECLARE [VARIABLE] имя_локальной_переменной
    <тип_данных> [{ = | DEFAULT} <значение>]; ]
```

```
[ DECLARE [VARIABLE] имя_курсора [SCROLL | NO SCROLL]  CURSOR
FOR (<запрос_SELECT>); ]
[ DECLARE FUNCTION имя_функции RETURNS <тип_данных>
AS <тело_функции> ]
[ DECLARE PROCEDURE имя_процедуры [(<список_входных_параметров>)]
[ RETURNS (<список_выходных_параметров>)]
AS <тело_процедуры>]
```

Необязательный список объявления (раздел деклараций) служит для объявления локальных переменных и курсоров, а также может содержать описания подпрограмм (процедур и функций). При этом, если используются вложенные процедуры и функции, следует помнить, что они не могут в своем теле содержать других вложенных процедур и функций. Также во вложенных процедурах и функциях нельзя напрямую обращаться к переменным, курсорам и другим подпрограммам основной процедуры.

Аналогично описанному выше синтаксису вложенные процедуры и функции могут быть объявлены в теле функций, триггеров и выполнимых блоков.

Блок операторов представляет собой обязательный раздел исполнения (раздел исполняемого кода), может содержать код для обработки исключений. В разделе исполняемого кода могут использоваться как обычные локальные переменные, определенные в разделе деклараций, так и входные и выходные параметры. Как уже было отмечено, в SQL-запросах Firebird локальные переменные, входные и выходные параметры должны использоваться с двоеточием перед именем переменной, чтобы отличать их от имен столбцов таблиц.

В MS SQL Server хранимые процедуры (а также функции, триггеры) могут быть созданы с использованием языковых средств T-SQL или с помощью сборок (*assembly*) в среде CLR (Common Language Runtime) платформы Microsoft .NET Framework [4]. В настоящем учебнике рассматривается только T-SQL.

В MS SQL Server имеется несколько типов ХП:

1) *системные ХП* предназначены для выполнения различных административных действий. Системные ХП имеют префикс *sp_*, хранятся в системной БД и могут быть вызваны в контексте любой другой БД;

2) *пользовательские ХП* реализуют какие-либо действия. Каждая ХП располагается в конкретной БД, где и выполняется;

3) *временные ХП* существуют лишь некоторое время, после чего автоматически уничтожаются сервером. Они делятся на локальные и глобальные.

Локальные временные ХП могут быть вызваны только из того соединения, в котором созданы. При создании такой процедуры ей необходимо дать имя, начинающееся с одного символа *#*. Как и все

временные объекты, ХП этого типа автоматически удаляются при отключении пользователя, перезапуске или остановке сервера.

Глобальные временные ХП доступны для любых соединений сервера, на котором имеется такая же процедура. Для ее определения достаточно дать ей имя, начинающееся с символов `##`. Удаляются эти процедуры при перезапуске или остановке сервера, а также при закрытии соединения, в контексте которого они были созданы.

Для создания ХП используется запрос формата [7]

```
CREATE { PROC | PROCEDURE } [имя_схемы.] имя_процедуры [ ;номер ]
    [ <список_параметров> ]
    [ WITH <опция_процедуры> [, <опция_процедуры> ]... ]
    [ FOR REPLICATION ]
    AS <тело_процедуры>
```

где

```
<список_параметров> ::= @параметр1 <тип_данных>
                        [ VARYING ] [ = <значение> ]
                        [ OUT | OUTPUT ] [READONLY]
                        [,@параметр2 <тип_данных> ...]
<опция_процедуры> ::=
    [ ENCRYPTION ]
    [ RECOMPILE ]
    [ EXECUTE AS 'контекст' ]
```

Используя префиксы `sp_`, `#`, `##`, создаваемую ХП можно определить в качестве системной или временной.

Номер в имени процедуры представляет собой необязательный целочисленный аргумент, используемый для группирования одноименных процедур. Не рекомендуется для использования.

Для передачи входных и выходных данных в ХП могут использоваться параметры, имена которых должны начинаться с символа `@`. Параметр является выходным, если объявляется с ключевым словом `OUTPUT` (или сокращенно `OUT`). Возвращающий табличное значение тип данных не может быть указан в качестве типа для выходного параметра процедуры.

Ключевое слово `VARYING` может применяться при описании выходного параметра типа `CURSOR` и определяет результирующее множество на выходе процедуры.

При объявлении параметра допускается задавать значение по умолчанию, которое должно быть константой или значением `NULL`. Константа может иметь вид шаблона:

```
CREATE PROCEDURE p1
    @LastName nvarchar(50) = N'Д %',
    @FirstName nvarchar(50) = N' %'
AS ...
```

Ключевое слово READONLY указывает, что значение параметра не может быть изменено внутри ХП.

При создании процедуры после ключевого слова WITH можно указать ряд дополнительных опций. Опция RECOMPILE предписывает системе создавать план выполнения ХП при каждом ее вызове, т. е. выполнять перекомпиляцию кода процедуры. Опция ENCRYPTION предписывает серверу выполнить шифрование кода ХП, что может обеспечить защиту от использования авторских алгоритмов, реализующих работу ХП. Опция EXECUTE AS определяет контекст безопасности, в котором должна быть выполнена процедура [7].

Опция FOR REPLICATION указывает, что процедура создается для репликации. В таком случае не допускается задание определения процедуры с параметрами.

Тело процедуры в MS SQL Server, как и в Firebird, состоит из раздела деклараций и раздела исполнения. Локальные переменные объявляются в соответствии с синтаксисом, приведенным ранее в подпараграфе 6.1.2. Формат объявления вложенных процедур и функций реализуется без использования оператора DECLARE и будет рассмотрен далее при изучении выполняемых процедур (подпараграф 6.2.4).

В СУБД Oracle хранимые процедуры создаются запросом вида [1, 2]

```
CREATE [OR REPLACE] PROCEDURE имя_процедуры
  [ (<список_параметров>) ]
{ IS | AS }
  [<список_объявления>]
BEGIN
  <блок_исполнения>
  [ EXCEPTION
    <блок_обработки_исключений>]
END [имя_процедуры]
```

где

```
<список_параметров> ::= параметр1 [ IN | OUT | IN OUT]
                        <тип_данных> DEFAULT <значение>
                        [, параметр2 [ IN | OUT | IN OUT]
                        <тип_данных> DEFAULT <значение>...]
```

При создании процедуры (или функции) в Oracle можно задать три типа параметров:

1) IN — входной параметр (по умолчанию), может ссылаться на процедуру или функцию, при этом значение параметра не может быть изменено процедурой или функцией;

2) OUT — выходной параметр, не может ссылаться на процедуру или функцию, но значение параметра может быть изменено процедурой или функцией;

3) IN OUT — одновременно входной и выходной параметр, может ссылаться на процедуру или функцию, и значение параметра может быть изменено процедурой или функцией.

Список объявления переменных в ХП реализуется в соответствии с синтаксисом, приведенным ранее в подпараграфе 6.1.2, не используя при этом оператор DECLARE. Используются ключевые слова IS или AS перед списком, которые эквивалентны всегда, кроме объявления явного курсора (в этом случае необходимо использовать AS).

Для изменения определения процедуры в различных СУБД используются различные запросы. В Firebird используются запросы

```
{ ALTER | RECREATE | CREATE OR ALTER } PROCEDURE имя_процедуры
  [(<список_входных_параметров>)]
  [RETURNS (<список_выходных_параметров>)]
  AS <тело_процедуры>
```

Если применяется запрос ALTER PROCEDURE, то имя_процедуры должно быть именем существующей процедуры. Это «мягкий» способ изменения кода процедуры, потому что, если у нее есть зависимости (процедура используется другими объектами БД, например другой ХП), на которые логически не влияют изменения, то они будут сохранены.

Запрос RECREATE PROCEDURE идентичен запросу CREATE PROCEDURE, за исключением того, что для существующей процедуры с тем же именем он сначала выполняет ее удаление перед созданием нового объекта. Следует учесть, что запрос RECREATE PROCEDURE не будет выполнен, если процедура используется другими объектами БД.

Запрос CREATE OR ALTER PROCEDURE создает процедуру, если она не существует, иначе — изменит определение существующей процедуры и перекомпилирует ее. При этом имеющиеся зависимости и привилегии сохраняются.

В MS SQL Server изменение определения ХП выполняется запросом ALTER PROCEDURE, синтаксис которого повторяет синтаксис запроса на создание процедуры:

```
ALTER { PROC | PROCEDURE } [имя_схемы.] имя_процедуры [ ; номер ]
  [ <список_параметров> ]
  [ WITH <опция_процедуры> [, <опция_процедуры> ]... ]
  [ FOR REPLICATION ]
  AS <тело_процедуры>
```

Другие запросы на изменение определения ХП в MS SQL Server не поддерживаются.

В Oracle ХП может быть изменена (полностью пересоздана) запросом CREATE OR REPLACE PROCEDURE либо перекомпилирована

с помощью запроса ALTER PROCEDURE. Запрос на перекомпиляцию процедуры имеет формат

```
ALTER PROCEDURE имя_процедуры  
COMPILE [DEBUG] [параметр_компиляции = <значение> [...]]  
[REUSE SETTINGS];
```

Опция COMPILE указывает на необходимость перекомпиляции процедуры. Опция DEBUG указывает компилятору на необходимость сгенерировать и сохранить код для использования PL/SQL-отладчиком. Во время перекомпиляции Oracle удаляет все заданные параметры компиляции [18], получает их вновь из сеанса и сохраняет их при окончании компиляции. Чтобы избежать этого процесса, используется опция REUSE SETTINGS.

Удаление хранимых процедур в любой из рассматриваемых СУБД осуществляется запросом DROP PROCEDURE. В Firebird и Oracle запрос имеет формат

```
DROP PROCEDURE имя_процедуры;
```

В MS SQL Server одним запросом можно удалить несколько ХП:

```
DROP PROCEDURE [имя_схемы.] имя_процедуры1 [,  
[имя_схемы.] имя_процедуры2 ...];
```

Удалить хранимую процедуру можно лишь в том случае, если ее не используют другие процедуры или триггеры.

6.2.2. Виды хранимых процедур

После того как ХП созданы в БД, их можно использовать для обработки данных и получения выходных результатов. Хранимые процедуры могут вызываться независимо — как с помощью утилиты для работы с БД, так и при выполнении SQL-сценария (в теле триггера и другой ХП). Кроме того, ХП могут быть *рекурсивными* — это означает, что из ХП можно вызвать саму себя.

В некоторых СУБД способы вызова различаются в зависимости от вида ХП.

Хранимые процедуры Firebird условно подразделяются на два типа: *селективные процедуры* (select procedure) и *выполняемые процедуры* (executable procedure). Каждая из этих процедур имеет разное назначение. Селективные ХП (процедуры выбора) используются для получения в качестве результата набора данных, состоящего из множества строк. Выполняемые ХП (процедуры действия) применяются для реализации сложных алгоритмов обработки данных и могут не возвращать набор строк либо возвращать однострочный результат в виде значений выходных параметров.

Создание ХП обоих типов формально не отличается, но у них разный вызов. Для вызова селективной процедуры используется запрос SELECT, выполняемая процедура вызывается с помощью оператора EXECUTE. При этом и селективные, и выполняемые процедуры могут быть рекурсивными.

Также отличаются операторы, которые влияют на ход выполнения ХП. В хранимых селективных процедурах применяется оператор SUSPEND. Он приостанавливает выполнение процедуры для возвращения из нее текущих значений выходных параметров, после чего выполнение процедуры продолжается. Если выходным параметрам не присвоены значения, то будут возвращены NULL. Оператор SUSPEND не должен использоваться в выполняемых процедурах, так как команды, следующие за ним, никогда не будут выполнены.

В выполняемых процедурах для управления ходом программы используются такие операторы, как BREAK, LEAVE, CONTINUE и EXIT. Все эти операторы, кроме EXIT, используются только внутри циклов.

Оператор BREAK предназначен для досрочного завершения цикла. При его выполнении происходит немедленный выход из текущего цикла и переход к выполнению инструкции, следующей сразу после ключевого слова END, обозначающего конец цикла. Часто, но не всегда, оператор BREAK встречается после проверки условия оператора ветвления IF.

Как в селективных ХП, так и в выполняемых процедурах может использоваться оператор LEAVE, который служит для выхода из текущего цикла, а также для перехода на вышележащий цикл с меткой.

Оператор LEAVE без явного указания метки означает выход из текущего цикла, например в теле ХП:

```
FOR SELECT RequestCD, COALESCE(ExecutionDate, 'Дата неизвестна')
  FROM Request
  INTO :Code, :Exec_Date
DO
BEGIN
  IF (Exec_Date = 'Дата неизвестна') THEN
    LEAVE; -- выход из цикла выборки курсора
  ELSE
    SUSPEND;
END
```

Оператор LEAVE с явным указанием метки позволяет приостановить выполнение текущего блока и перейти на вышележащий цикл, помеченный соответствующей меткой. В этом случае используется синтаксис

```
[имя_метки:]
{FOR | WHILE} DO
BEGIN
```

```

...
LEAVE [имя_метки];
...
END

```

Оператор цикла представляет собой оператор WHILE, неявный курсор или оператор EXECUTE STATEMENT с циклом FOR.

Переход на метку с помощью LEAVE <имя_метки> осуществляется обычно по какому-то определенному условию, т. е. оператор LEAVE имя_метки используется после фразы THEN или ELSE оператора ветвления IF. Причем оператор перехода LEAVE имя_метки может быть вызван из цикла, вложенного в тот цикл, на который осуществляется переход.

Помимо оператора LEAVE, в циклах FOR и WHILE может также использоваться оператор CONTINUE, который позволяет прервать выполнение текущей итерации цикла и начать новую итерацию.

Для досрочного завершения выполнения ХП может применяться оператор EXIT. Вызов EXIT в разделе исполняемого кода процедуры вызывает переход на конечный оператор END в процедуре и используется для того, чтобы прервать выполнение ХП и вернуться в точку ее вызова, т. е. приводит к немедленному завершению ХП.

Помимо операторов прерывания и перехода, в ХП Firebird могут использоваться и другие переменные и операторы.

Для получения количества строк, возвращаемых выполненным запросом DML или запросом SELECT, предназначена контекстная переменная ROW_COUNT типа INTEGER. Она должна использоваться в том же блоке, что и запрос, например внутри тела ХП:

```

UPDATE Abonent SET Fio = :Fio WHERE AccountCD = :Account;
IF (ROW_COUNT = 0) THEN
  INSERT INTO Abonent (AccountCD, Fio) VALUES (:Account, :Fio);

```

В данном примере выполняется запрос на обновление в таблице Abonent столбца Fio новым значением для абонента с заданным номером лицевого счета (:Account номер лицевого счета и :Fio ФИО — локальные переменные). Если строки с заданным номером лицевого счета нет в таблице, то контекстная переменная ROW_COUNT принимает значение ноль и далее выполняется вставка строки в таблицу с данными номером лицевого счета и ФИО.

Контекстная переменная ROW_COUNT также может применяться в цикле выборки (FETCH) при использовании явного курсора, как уже было показано ранее при изучении курсоров.

Примечание. Стандартные запросы языка определения данных (DDL) в модулях на процедурном языке можно выполнять только с помощью специального оператора (EXECUTE, EXECUTE STATEMENT).

Следует отметить еще одну важную особенность, которой можно пользоваться в ХП и других процедурных объектах. В гл. 8 подробно рассмотрены транзакции в БД, однако уже здесь при описании процедурного языка необходимо упомянуть оператор `IN AUTONOMOUS TRANSACTION DO`, который позволяет выполнить некоторый блок процедурного кода в автономной транзакции. Основной областью применения данного оператора могут служить те случаи, когда в основной транзакции необходимо вызвать исключение (или оно может возникнуть в связи с ошибкой), но при этом необходимо, чтобы определенные изменения в БД (некоторый блок кода) были в любом случае сохранены. В этом случае такой блок процедурного кода необходимо выполнять в автономной транзакции. Пример использования оператора `IN AUTONOMOUS TRANSACTION DO` приведен в подпараграфе 6.6.2 при рассмотрении триггеров DDL.

Система управления БД MS SQL Server, в отличие от Firebird, не поддерживает селективные ХП. Это означает, что в запросе `SELECT` использовать хранимую процедуру не получится, MS SQL Server такой функциональности не предоставляет. Фактически все процедуры, создаваемые пользователем в MS SQL Server, являются выполняемыми процедурами.

Для возврата данных вызывающей клиентской программе в процедурах MS SQL Server используются выходные параметры и коды возврата.

Выходные параметры обычно используют, когда из хранимой процедуры требуется получить одно значение или одну строку информации, а не набор данных. Для возвращения одной строки информации гораздо выгоднее в смысле быстродействия использовать выходные параметры, чем подготавливать набор данных.

В случае, если все-таки необходимо получить из ХП набор данных, используют несколько вариантов обхода проблемы невозможности использования ХП в запросе `SELECT` [7]:

- 1) оформить хранимую процедуру как функцию. Функции в MS SQL Server допустимо использовать в секции `FROM` запроса `SELECT`. Однако в силу ряда ограничений, накладываемых на синтаксис пользовательских функций (см. далее параграф 6.3), переписать хранимую процедуру в функцию иногда просто невозможно;

- 2) получить набор данных через временную таблицу. При этом следует учесть, что так можно получить только один не вложенный набор данных. При попытке получить таким образом набор из ХП, которая, в свою очередь, получает набор из другой процедуры, возникнет ошибка «`INSERT EXEC statement cannot be nested`»;

- 3) получить многострочный результат за счет использования выходного параметра типа `CURSOR`. При этом следует учесть, что курсоры в MS SQL Server работают довольно медленно за счет по-

строчной обработки данных, поэтому для больших выборок такой способ плохо подходит.

При именовании пользовательских ХП не следует использовать `sp_` в качестве начальной части имени, так как с этих символов начинаются имена системных ХП MS SQL Server.

Для управления ходом выполнения процедуры могут использоваться операторы `BREAK`, `GOTO`, `CONTINUE` и `RETURN`.

Операторы `BREAK` и `CONTINUE` аналогичны одноименным операторам в Firebird. Оператор `GOTO` аналогичен оператору `LEAVE` в Firebird и позволяет организовать переход по указанной метке.

Примечание. Не рекомендуется использовать в программах операторы перехода по меткам. При грамотном использовании операторов ветвления и циклов потребность в использовании операторов перехода не существует.

Процедура может возвращать целочисленное значение, называемое кодом возврата, с помощью оператора `RETURN`. Оператор `RETURN` используется в формате

```
RETURN [ <код_возврата> ];
```

где `<код_возврата>` — целое число (по умолчанию 0), которое может использоваться в вызывающей программе для анализа результатов выполнения ХП. При вызове оператор осуществляет выход из ХП (пользовательской функции, триггера).

Для вызова хранимой процедуры используется оператор `EXECUTE` (допустимо сокращение `EXEC`).

Для получения количества строк, возвращаемых выполненным запросом, в MS SQL Server может использоваться контекстная переменная `@@ROWCOUNT`, аналогичная переменной `ROW_COUNT` в Firebird.

Система управления БД Oracle, как и MS SQL Server, не поддерживает селективные ХП, и все процедуры в Oracle являются выполняемыми процедурами действия.

Для вызова ХП используется оператор `EXECUTE` (или `EXEC`), также может применяться оператор `CALL`.

Для управления ходом выполнения процедур предусмотрены только операторы `EXIT` и `GOTO`. Оператор `EXIT` осуществляет безусловный выход из цикла и применяется совместно с оператором ветвления `IF`. Выход при выполнении условия может быть реализован при использовании оператора `EXIT WHEN`. Оператор `GOTO` реализует выход из цикла во внешний контекст. В PL/SQL нет такого, иногда полезного, оператора, как `CONTINUE`, вследствие того, что выражение `CONTINUE` зарезервировано языком PL/SQL и используется для дру-

гих целей. Но вызов CONTINUE можно эмулировать, применив цикл вида LOOP EXIT WHEN END LOOP и используя весьма непопулярный, но в данном случае очень полезный оператор GOTO.

Количество строк, возвращаемых последним выполненным запросом DML или SELECT, доступно через переменную SQL%ROWCOUNT.

6.2.3. Селективные процедуры Firebird

Селективные процедуры возвращают результирующий набор строк, включающих столбцы, выбранные из одной или нескольких таблиц или представлений. Использование селективных процедур позволяет объединять данные, получаемые несколькими запросами, и представлять полученные данные в виде единого набора выходных параметров селективной процедуры.

Например, пусть стоит задача получения какой-то информации из таблиц БД и для этой цели используется запрос с группировкой по какому-то полю внешнего ключа, представляющего собой код, значение которого расшифровывается в другой таблице. В запросе с группировкой не удастся включить в список возвращаемых элементов расшифровку значения кода внешнего ключа из другой таблицы (например, название улицы по внешнему ключу в таблице абонентов учебной БД). Если использовать в ХП два последовательно выполняемых запроса — один на группировку, другой на расшифровку кода, а затем возвращать полученные результаты в виде одного набора данных, то, во-первых, можно упростить получение какой-либо аналитической информации (для данного примера) и, во-вторых, оптимизировать этот процесс за счет использования ХП.

Для выполнения селективной процедуры используется запрос SELECT, в секции FROM которого указывается имя ХП. Синтаксис запроса SELECT в этом случае имеет вид

```
SELECT <список_возвращаемых_элементов>  
FROM имя_процедуры [( [ входной_параметр1 [, входной_параметр2 ... ] ] )]  
[ WHERE <условие_поиска> ]  
[ ORDER BY <элемент_сортировки1> [, <элемент_сортировки2> ] ... ];
```

Рассмотрим следующий SQL-сценарий, который создает селективную хранимую процедуру с именем ListAbonent в учебной БД Firebird:

```
CREATE PROCEDURE ListAbonent  
RETURNS (LAccountCD VARCHAR(6), LFio VARCHAR(20),  
         LPayDate DATE, LPaySum NUMERIC(15,2),  
         LPayMonth SMALLINT, LPayYear SMALLINT)  
AS BEGIN  
FOR SELECT A.AccountCD, A.Fio, P.PayDate,  
         P.PaySum, P.PayMonth, P.PayYear  
FROM Abonent A, PaySumma P
```

```

WHERE A.AccountCD = P.AccountCD AND P.PaySum > 450
INTO :LAccountCD, :LFio, :LPayDate, :LPaySum,
      :LPayMonth, :LPayYear
DO
SUSPEND;
END;

```

Процедура ListAbonent в выходных параметрах возвращает строку, состоящую из номера лицевого счета абонента (LAccountCD), ФИО абонента (LFio), даты оплаты (LPayDate), суммы оплаты (LPaySum), месяца (LPayMonth) и года (LPayYear), за которые производится оплата, с суммой, большей 450.

Оператор SUSPEND приостанавливает выполнение процедуры ListAbonent для возвращения из нее текущих значений выходных переменных, после чего выполнение процедуры продолжается.

В процедуре ListAbonent производятся следующие действия:

1) запрос SELECT формирует соединение таблиц Abonent, PaySumma и возвращает НД;

2) курсор FOR SELECT ... DO для каждой строки таблицы результатов, сформированной запросом SELECT, выполняет блок операторов, который следует за секцией DO;

3) в данном случае выполняется один оператор SUSPEND. Он приостанавливает выполнение процедуры, чтобы передать через выходные параметры процедуры очередную строку из НД.

Чтобы выбрать все значения, возвращаемые селективной процедурой ListAbonent, необходимо выполнить запрос

```
SELECT * FROM ListAbonent;
```

Результат выполнения запроса к ХП ListAbonent представлен на рис. 6.5.

Результат, возвращаемый данной процедурой, может быть легко получен с помощью запроса SELECT без оформления ХП. Однако использование даже такой несложной процедуры вместо обычного запроса SELECT дает ряд преимуществ (простота доступа, сокращение объема кода клиентского приложения и другие, описанные ранее), не говоря уже о процедурах, реализующих более сложные алгоритмы.

Следующий запрос демонстрирует возможность вызова селективной ХП в секции FROM запроса, на котором основано определяемое представление:

```
CREATE VIEW P_Listab AS SELECT * FROM ListAbonent;
```

Результат выполнения запроса

```
SELECT * FROM P_Listab;
```

совпадает с результатом, представленным на рис. 6.5.

LACCOUNTCD	LFIO	LPAYDATE	LPAYSUM	LPAYMONTH	LPAYYEAR
115705	Мищенко Е. В.	02.02.2017	553,85	1	2017
015527	Конюхов В. С.	08.08.2017	580,10	7	2017
015527	Конюхов В. С.	03.11.2018	611,30	10	2018
443690	Тулупова М. И.	05.09.2016	485,00	8	2016
443069	Стародубцев Е. В.	03.06.2017	466,69	5	2017
136169	Денисова Е. К.	26.11.2018	528,44	10	2018
080270	Тимошкина Н. Г.	12.05.2018	460,00	4	2018

Рис. 6.5. Результат выполнения запроса к процедуре ListAbonent

Рассмотрим пример SQL-сценария для изменения определения процедуры ListAbonent для добавления входных параметров Mes и God. В теле следующей процедуры входные параметры Mes и God используются для отбора строк, содержащих информацию об оплатах с суммой больше 500, произведенных абонентом за заданный месяц (Mes) указанного года (God):

```
ALTER PROCEDURE ListAbonent (Mes SMALLINT, God SMALLINT)
    RETURNS (LAccountCD VARCHAR(6), LFio VARCHAR(20),
            LPayDate DATE, LPaySum NUMERIC(15,2),
            LPayMonth SMALLINT, LPayYear SMALLINT)
AS BEGIN
    FOR SELECT A.AccountCD, A.Fio, P.PayDate,
            P.PaySum, P.PayMonth, P.PayYear
    FROM Abonent A, PaySumma P
    WHERE A. AccountCD = P.AccountCD
        AND P.PaySum > 500 AND :Mes = P.PayMonth AND :God = P.PayYear
    INTO :LAccountCD, :LFio, : LPayDate,
        :LPaySum, :LPayMonth, :LPayYear
    DO
        SUSPEND;
END;
```

Если селективная процедура определена с указанием входных параметров, то их задание при вызове процедуры в запросе SELECT является обязательным. Например, чтобы вывести все сведения о значениях оплат, больших 500, за октябрь 2018 г., необходимо выполнить запрос

```
SELECT * FROM ListAbonent (10, 2018);
```

Результат представлен на рис. 6.6.

Рассмотрим пример решения задачи получения информации о начислениях, оплатах и остатках у заданного абонента по заданной услуге за определенный период с детализацией по месяцам. Под остатком на конец заданного периода будем понимать остаток на начало этого периода плюс значение начисления за этот период и минус значение оплаты за этот же период. Остаток на начало заданного периода определяется разницей между суммами всех начислений и оплат за период, предшествующий заданному. Скрипт для создания процедуры Oborot_Saldo_Vedomost, отвечающей за вывод требуемых данных, может выглядеть так:

```
CREATE PROCEDURE Oborot_Saldo_Vedomost (Account VARCHAR(6),
    ServiceCD INTEGER, NMonth TMonth, NYear TYear,
    FMonth TMonth, Fyear TYear)
    RETURNS (NachislMonth TMonth, NachislYear TYear, Begin_Ostatok Money,
            Nachisl Money, Pay Money, Result_Ostatok Money) AS
    DECLARE VARIABLE i INTEGER = 1;
```

LACCOUNTCD	LFIO	LPAYDATE	LPAYSUM	LPAYMONTH	LPAYYEAR
015527	Конюхов В. С.	03.11.2018	611,30	10	2018
136169	Денисова Е. К.	26.11.2018	528,44	10	2018

Рис. 6.6. Результат выполнения запроса к модифицированной процедуре ListAbonent

```

BEGIN
  SELECT
    /* вычисление остатка на начало 1-го месяца периода */
    /* сумма начислений на начало 1-го месяца периода */
    (COALESCE((SELECT SUM(N.NachislSum) SumNach FROM NachislSumma N
      WHERE N.AccountCD = :Account
      AND N.ServiceCD = :ServiceCD
      AND (N.NachislYear < :NYear
        OR (N.NachislYear = :NYear
          AND N.NachislMonth < :Nmonth))),0) -
    /* сумма оплат на начало 1-го месяца периода */
    COALESCE ((SELECT SUM(P.PaySum) SumPay FROM PaySumma P
      WHERE P.AccountCD = :Account
      AND P.ServiceCD = : ServiceCD
      AND (P.PayYear < :NYear
        OR (P.PayYear = :NYear
          AND P.PayMonth < :NMonth))),0))
  FROM RDB$DATABASE INTO :Begin_Ostatok;
  WHILE (i > 0) DO
  BEGIN
    SELECT
      /* начисление за месяц */
      COALESCE ((SELECT Na.NachislSum FROM NachislSumma Na
        WHERE Na.AccountCD = :Account
        AND Na.ServiceCD = :ServiceCD
        AND Na.NachislYear = :NYear
        AND Na.NachislMonth = :NMonth),0),
      /* оплата за месяц */
      COALESCE ((SELECT Pa.PaySum FROM PaySumma Pa
        WHERE Pa.AccountCD = :Account
        AND Pa.ServiceCD = :ServiceCD
        AND Pa.PayYear = :NYear
        AND Pa.PayMonth = :NMonth),0)
    FROM RDB$DATABASE INTO :Nachisl, :Pay;
    /* вычисление остатка на конец месяца */
    Result_Ostatok = Begin_Ostatok + (Nachisl - Pay);
    NachislMonth = NMonth;
    NachislYear = NYear;
    SUSPEND;
    Begin_Ostatok = Result_Ostatok;
    IF (NMonth = FMonth AND NYear = FYear) THEN EXIT;
    IF (NMonth = 12)
      THEN BEGIN
        NYear = NYear + 1;
        NMonth = 1;
      END
      ELSE NMonth = NMonth + 1;
  END
END
END;

```

Процедура `Oborot_Saldo_Vedomost` для определенного абонента (Account) по указанной услуге (ServiceCD) по каждому месяцу,

попадающему в заданный период (от месяца NMonth в NYear году до месяца FMonth в FYear году), выводит:

- 1) значение денежного остатка на начало месяца (Begin_Ostatok);
- 2) значение начисления за месяц (Nachisl);
- 3) значение оплаты за месяц (Pay);
- 4) значение денежного остатка на конец месяца (Result_Ostatok).

В секции FROM запроса SELECT указана системная таблица RDB\$DATABASE, которая используется в качестве «фиктивной» таблицы. Использование в данном контексте подобной таблицы позволяет избежать ошибок, связанных с «размножением» одних и тех же данных.

При каждом проходе цикла запрос возвращает одну строку с информацией за очередной месяц. Информация выводится по каждому месяцу анализируемого периода, а не только по тем месяцам, за которые есть начисление или оплата в таблицах соответственно NachislSumma или PaySumma. В цикле также происходит переход к следующему месяцу. Работа процедуры заканчивается после того, как будет выведена информация за последний месяц заданного периода (месяц FMonth, год FYear).

Например, чтобы вывести информацию о начислениях, оплатах и об остатках у абонента с номером лицевого счета '015527' по услуге с кодом 1 за период с января 2017 г. по декабрь 2017 г., необходимо выполнить запрос

```
SELECT * FROM Oborot_Saldo_Vedomost ('015527',1,1,2017,12,2017);
```

Результат выполнения запроса представлен на рис. 6.7.

Рассмотрим пример селективной процедуры, в которой используется обновляемый явный курсор для позиционированного удаления и обновления строк таблицы. Создадим процедуру Exec_Req, которая путем позиционированной модификации строк таблицы Request удаляет все невыполненные ремонтные заявки, а все непогашенные ремонтные заявки преобразует в погашенные. При этом в таблицу Executor заносится дополнительная информация по исполнителям, для которых последним произведено удаление или погашение заявок. Соответствующий скрипт выглядит так:

```
ALTER TABLE Executor Add Info VARCHAR(40);
CREATE PROCEDURE Exec_Req
RETURNS (ECode INTEGER, EName VARCHAR(20), Info VARCHAR(40))
AS
DECLARE EDate DATE;
DECLARE Exec BOOLEAN;
DECLARE Req CURSOR FOR
(SELECT ExecutorCD, ExecutionDate, Executed FROM Request);
BEGIN
```

NachisMonth	NachisYear	Begin_Ostatok	Nachis1	Pay	Result_Ostatok
1	2017	0,00	0,00	0,00	0,00
2	2017	0,00	18,32	20,00	-1,68
3	2017	-1,68	0,00	0,00	-1,68
4	2017	-1,68	0,00	0,00	-1,68
5	2017	-1,68	0,00	0,00	-1,68
6	2017	-1,68	0,00	0,00	-1,68
7	2017	-1,68	28,32	30,00	-3,36
8	2017	-3,36	0,00	0,00	-3,36
9	2017	-3,36	0,00	0,00	-3,36
10	2017	-3,36	0,00	0,00	-3,36
11	2017	-3,36	0,00	0,00	-3,36
12	2017	-3,36	0,00	0,00	-3,36

Рис. 6.7. Результат выполнения запроса к процедуре **Oborot_Saldo_Vedomost**

```

OPEN Req;
WHILE (1 = 1) DO
BEGIN
    FETCH Req INTO :ECode, :EDate, :Exec;
    IF (ROW_COUNT = 0) THEN LEAVE;
    IF (EDate IS NULL) THEN
        BEGIN
            DELETE FROM Request WHERE CURRENT OF Req;
            UPDATE Executor SET Info = 'DEL_NotExec'
            WHERE ExecutorCD = :ECode;
        END
    ELSE
        IF (NOT Exec) THEN
            BEGIN
                UPDATE Request SET Executed = TRUE WHERE CURRENT OF Req;
                UPDATE Executor SET Info = 'UPD_Executed'
                WHERE ExecutorCD = :ECode
            END
        END
    END
CLOSE Req;
FOR
    SELECT ExecutorCD, Fio, Info
    FROM Executor
    INTO :ECode, :EName, :Info
    DO
        SUSPEND;
END;

```

В начале скрипта с помощью запроса ALTER TABLE в таблицу Executor добавляется столбец Info. После этого создается процедура Exec_Req, в которой сначала явным курсором Req выполняется выборка строк из таблицы Request, а затем проверяется ряд условий.

Если очередная ремонтная заявка не выполнена (EDate IS NULL), то происходит ее позиционированное удаление, а в таблице Executor столбец Info получает значение 'DEL_NotExec' (признак того, что для данного исполнителя удалены невыполненные заявки).

Если выполненная ремонтная заявка не погашена (NOT Exec), то происходит позиционированное обновление (Executed = TRUE) той строки таблицы Request, на которой в данный момент находится курсор Req. Затем в таблице Executor происходит обновление столбца Info. Столбец Info получает значение 'UPD_Executed' (признак того, что для данного исполнителя непогашенные ранее заявки погашены).

Если очередная ремонтная заявка и выполнена, и погашена, то позиционированная модификация строки в таблице Request не выполняется и для исполнителя соответствующей заявки в таблице Executor столбец Info не обновляется (будет иметь NULL).

Затем курсор Req закрывается, и происходит выборка в цикле FOR SELECT ... DO (неявный курсор) всех столбцов таблицы Executor в выходные параметры процедуры Exec_Req.

Процедура Exec_Req выполняется с помощью запроса

```
SELECT * FROM Exec_Req;
```

Результат выполнения запроса представлен на рис. 6.8.

ECODE	ENAME	INFO
1	Стародубцев Е. М.	UPD_Executed
2	Булгаков Т. И.	<null>
3	Шубин В. Г.	UPD_Executed
4	Шлюков М. К.	DEL_NotExec
5	Школьников С. М.	<null>

Рис. 6.8. Результат работы процедуры Exec_Req

В таблице Request будут удалены две строки, где значение столбца ExecutionDate равно NULL (ремонтные заявки с кодами 5 и 16), а в строках с кодами заявок 3 и 10 значение столбца Executed примет значение TRUE.

Рассмотрим пример селективной процедуры, в которой объявлены и используются вложенные процедуры. Создадим процедуру Information, которая по заданному номеру лицевого счета выводит информацию об абоненте и его ремонтных заявках. При этом процедура использует две вложенные процедуры: AbonentInformation, которая выводит ФИО, адрес и телефон абонента, и RequestInformation, которая выводит дату поступления, дату выполнения и признак погашения заявки абонента. Скрипт для создания процедуры:

```
CREATE PROCEDURE Information (AccountCD VARCHAR(6))
RETURNS (FIO VARCHAR(20),Address VARCHAR(50),Phone VARCHAR(15),
        IncomingDate DATE,ExecutionDate DATE,Executed BOOLEAN)
AS
  DECLARE VARIABLE i INTEGER;
  DECLARE VARIABLE f DOUBLE PRECISION;
  DECLARE PROCEDURE AbonentInformation (AccountCD VARCHAR(6))
  RETURNS (FIO VARCHAR(20),Address VARCHAR(50),Phone VARCHAR(15))
  AS
  BEGIN
    FOR SELECT A.Fio,
      (SELECT S.StreetNM FROM Street S
       WHERE S.StreetCD = A.StreetCD)
      ||', д.'||HouseNo||', кв.'||FlatNo AS Adress,
      A.Phone
    FROM Abonent A
    WHERE A.AccountCD = :AccountCD
    INTO FIO,Address,Phone
  DO
```



```

        SUSPEND;
    END
    DECLARE PROCEDURE RequestInformation (AccountCD VARCHAR(6))
    RETURNS (IncomingDate DATE, ExecutionDate DATE, Executed BOOLEAN)
    AS
    BEGIN
        FOR SELECT R.IncomingDate,
                    R.ExecutionDate,
                    R.Executed
        FROM Request R
        WHERE R.AccountCD = :AccountCD
        INTO IncomingDate, ExecutionDate, Executed
        DO
            SUSPEND;
        END
    BEGIN
        FOR SELECT A1.*, RI.*
        FROM AbonentInformation(:AccountCD) AI,
            RequestInformation(:AccountCD) RI
        INTO FIO, Address, Phone, IncomingDate, ExecutionDate, Executed
        DO
            SUSPEND;
        END;
    END;

```

Примером выполнения разработанных процедур может быть следующий запрос:

```
SELECT * FROM Information('005488');
```

В заключение приведем текст SQL-сценария по созданию селективной ХП, осуществляющей проверку того, является ли входная строка вещественным числом:

```

CREATE PROCEDURE Istr(Str VARCHAR(255)) RETURNS (Ir FLOAT)
AS
BEGIN
    IF (Str SIMILAR TO
        '([\+|-]?[0-9]*([0-9].|[0-9])?[:DIGIT:])*' ESCAPE '\') THEN
        BEGIN
            Ir = Str;
            SUSPEND;
        END
    ELSE
        EXCEPTION Ins_Restrict 'Не число';
    WHEN SQLCODE -413 DO
        EXCEPTION Ins_Restrict 'Пустая строка';
    END

```

Проверка входной строки на число осуществляется с помощью регулярного выражения. Процедура возвращает вещественное чис-

ло или сообщает путем вызова исключения Ins_Restrict, что входная строка не является числом. Оператор WHEN обрабатывает ошибку при задании пустой строки в качестве параметра (код ошибки равен -413).

Примеры вызова процедуры Istr:

```
SELECT * FROM Istr('25');
SELECT * FROM Istr('25.54');
SELECT * FROM Istr('-25.5');
SELECT * FROM Istr(+123.45);
SELECT * FROM Istr('SQL');
SELECT * FROM Istr('');
```

6.2.4. Выполняемые процедуры

Выполняемые процедуры производят некоторые действия и могут, в общем случае, не возвращать результирующий набор строк, как это происходит в селективных процедурах. Если выполняемая процедура возвращает набор строк, то такая процедура фактически является *процедурой выбора*. В любом случае выполняемые процедуры не могут быть вызваны с помощью запроса SELECT (не являются селективными), для их вызова применяются специальные операторы. Все рассматриваемые в учебнике СУБД поддерживают выполняемые ХП в той или иной форме.

В Firebird синтаксис определения выполняемых процедур аналогичен синтаксису определения селективных процедур с тем исключением, что выполняемые процедуры могут не содержать выходных параметров, а для правильной работы селективной процедуры наличие выходных параметров обязательно. Запрос на вызов выполняемых хранимых процедур в Firebird имеет формат

```
EXECUTE PROCEDURE имя_процедуры [ ( ) [входной_параметр1]
                                     [, входной_параметр2 ...] [ ) ]
[RETURNING_VALUES выходной_параметр1 [,выходной_параметр2 ...]];
```

В качестве входных параметров, как и при вызове селективной процедуры, могут использоваться выражения.

Секция RETURNING_VALUES предназначена для присвоения ка-ким-либо переменным значений, возвращаемых вызываемой процедурой.

Примечание. Через EXECUTE PROCEDURE можно вызывать и селективные процедуры, если требуется получить только одно, первое значение.

В MS SQL Server для вызова ХП используется оператор EXECUTE (или, сокращенно EXEC) в формате

```
[ { EXEC | EXECUTE } ]
[ @имя_переменной_возврата = ] имя_процедуры [ ;номер]
[ [ @параметр1 = ] { <значение> | @имя_переменной [ OUTPUT ]
| [ DEFAULT ] } ] [ ,... ]
[ WITH RECOMPILE ];
```

В операторе ключевое слово EXECUTE должно использоваться обязательно, если вызов ХП не является единственной командой в пакете либо если процедура вызывается из тела другой процедуры или триггера.

Если процедура возвращает целочисленное значение с помощью оператора RETURN, то в операторе EXECUTE перед именем процедуры нужно записать имя локальной переменной (имя_переменной_возврата), которой будет присвоено возвращаемое значение, и знак равенства. Большинство системных процедур имеют возвращаемые значения, но, поскольку они являются параметрами по умолчанию, их можно игнорировать в описании вызова процедуры.

После имени вызываемой процедуры указывается список значений передаваемых процедуре параметров. Если параметр является выходным, то он задается в виде имени локальной переменной, за которым должно идти ключевое слово OUTPUT. Если при вызове процедуры для параметра указывается ключевое слово DEFAULT, то будет использовано значение по умолчанию.

Из синтаксиса оператора EXECUTE следует, что имена параметров могут быть опущены при вызове процедуры. Однако в этом случае пользователь должен указывать значения для параметров в том же порядке, в каком они перечислялись при создании процедуры. Присвоить параметру значение по умолчанию, просто пропустив его при перечислении, нельзя. Если же требуется опустить параметры, для которых определено значение по умолчанию, то достаточно явного указания имен параметров при вызове ХП. Более того, таким способом можно перечислять параметры и их значения в произвольном порядке.

Следует учесть, что при вызове процедуры указываются либо имена параметров со значениями, либо только значения без имени параметра. Их комбинирование не допускается.

Система управления БД Oracle для вызова ХП использует оператор EXECUTE (EXEC) либо CALL. Применяется формат

```
{ EXEC | EXECUTE | CALL } имя_процедуры
[ ( ] [ [параметр1 =>] <значение> ]
[ , [параметр2 =>] <значение>... ] [ ) ];
```

При вызове процедуры в Oracle доступно позиционное (без указания имени) либо именованное (с указанием имени, через знак =>) присвоение значений параметрам. Выходные параметры задаются

в виде имен локальных переменных, объявленных перед вызовом процедуры, в которые будет возвращен результат выполнения ХП.

Рассмотрим примеры работы с выполняемыми процедурами в различных СУБД. В подпараграфе 6.2.3 были описаны селективные процедуры Firebird и вывод данных с их помощью. Системы управления БД MS SQL Server и Oracle, как уже отмечалось, не поддерживают селективные ХП, все реализуемые в них процедуры являются выполняемыми. Однако это не означает, что в данных СУБД с помощью хранимых процедур нельзя организовать вывод информации из БД.

Пусть, например, необходимо вывести с помощью ХП среднее значение оплат по указанному лицевому счету, увеличенное в заданное число раз. Для решения задачи создадим процедуру SummaryPay и на этом примере продемонстрируем стандартное создание и использование ХП с входными и выходными параметрами. Скрипт создания процедуры в MS SQL Server:

```
CREATE PROCEDURE SummaryPay
    @LAccountCD NVARCHAR(6), @Koeff INT, @ResultSum MONEY OUTPUT
AS
    SET NOCOUNT ON;
    SELECT @ResultSum = AVG(PaySum)*@Koeff
    FROM PaySumma
    WHERE AccountCD = @LAccountCD;
GO
```

В приведенной процедуре оператор SET NOCOUNT ON запрещает вывод количества строк всеми инструкциями ХП, что может значительно повысить производительность за счет снижения объема сетевого трафика [7]. Запрос SELECT внутри процедуры использует значение, передаваемое входному параметру @LAccountCD, для получения среднего значения оплат абонента и после умножения этого значения на входящий коэффициент @Koeff присваивает вычисленное значение выходному параметру @ResultSum.

Выполним созданную процедуру с помощью следующего кода:

```
DECLARE @vResultSum MONEY;
EXECUTE SummaryPay '015527', 2, @vResultSum OUTPUT;
SELECT @vResultSum;
```

В приведенном коде сначала объявляется локальная переменная @vResultSum, в которую будет сохраняться значение выходного параметра @ResultSum процедуры. Затем вызывается процедура с указанием значений входных параметров и сохранением выходного параметра в локальную переменную. В операторе EXECUTE имена параметров опущены при вызове процедуры, т. е. выполняется

позиционное присвоение значений параметрам. Вывод результата (542,1333) осуществляется с помощью запроса SELECT.

Примечание. Имена выходных параметров ХП и локальных переменных, принимающих результаты, могут быть разными, однако типы данных и порядок расположения должны совпадать.

В общем случае значения могут быть указаны не только для входных параметров, но также и для параметров OUTPUT. Это позволяет ХП получать значение из вызываемой программы, изменять его или выполнять операции с этим значением, а затем возвращать новое значение вызываемой программе.

Например, переменной @vResultSum может быть присвоено значение прежде, чем программа вызовет процедуру SummaryPay. В таком случае при вызове процедуры будет передано значение переменной @vResultSum выходному параметру @ResultSum. Далее в тексте процедуры значение можно использовать для вычислений, формирующих новое значение. Оно передается обратно из процедуры через выходной параметр, обновляя значение в переменной @vResultSum при завершении процедуры. Часто это называется «возможностью передачи по ссылке».

Система управления БД Oracle позволяет создавать ХП с выходными параметрами, однако необходимо учитывать особенности их использования. Выходные параметры, возвращаемые процедурами, могут использоваться для различных вычислений внутри PL/SQL кода или как входные параметры для других процедур, однако вывести из PL/SQL значения выходных параметров на экран с помощью, например, обычного запроса SELECT нельзя.

Самый быстрый способ в Oracle вывести результаты на экран — использование специального пакета DBMS_OUTPUT, входящего в состав PL/SQL. Принцип работы пакета DBMS_OUTPUT заключается в открытии внутреннего буфера заданного размера, записи и (или) чтении нужных значений и закрытии буфера.

Основными операциями при работе с пакетом DBMS_OUTPUT являются следующие.

1. Операции управления буфером: открытие буфера заданного размера (buffer_size) и закрытие передачи в буфер. Это позволяют делать процедуры пакета ENABLE и DISABLE соответственно, вызов которых выполняется в формате

```
DBMS_OUTPUT.ENABLE(buffer_size);  
DBMS_OUTPUT.DISABLE;
```

2. Запись значений в буфер — операция PUT. Выполнение операции PUT обеспечивается процедурами PUT, PUT_LINE и NEW_LINE. Процедуры PUT и PUT_LINE имеют следующий синтаксис:

```
DBMS_OUTPUT.{ PUT | PUT_LINE} ( a...z );
```

где аргументы а...z должны быть явно преобразованы к какому-то одному из трех типов данных: VARCHAR2, NUMBER или DATE.

Процедура PUT_LINE, в отличие от PUT, добавляет к аргументу символ новой строки. Вызов PUT_LINE аналогичен вызову PUT с последующим вызовом процедуры NEW_LINE. Процедура DBMS_OUTPUT.NEW_LINE проставляет конец строки.

3. Чтение значений из буфера — операция GET. Выполнение операции GET обеспечивается процедурами GET_LINE и GET_LINES. Процедура GET_LINE имеет синтаксис

```
DBMS_OUTPUT.GET_LINE  
(LINE OUT VARCHAR2, STATUS OUT INTEGER);
```

где LINE — одна строка буфера, а STATUS указывает на то, успешно или нет была считана эта строка.

Процедура GET_LINES определена в формате

```
DBMS_OUTPUT.GET_LINES  
(LINES OUT DBMSOUTPUT_LINESARRAY, NUMLINES IN OUT INTEGER)
```

где LINES — содержимое нескольких строк буфера, запрошенных аргументом NUMLINES.

Ознакомиться подробнее с методами пакета DBMS_OUTPUT и особенностями использования можно в работах [1, 2].

Другой распространенный способ вывода данных на экран — сохранить значения выходных параметров в некоторой таблице и затем, используя запрос SELECT, вывести данные из этой таблицы.

Сохранение и вывод данных с использованием промежуточной таблицы, а также вывод информации на экран с помощью пакета DBMS_OUTPUT, будут продемонстрированы в приведенных далее примерах.

Процедура SummaryPay в приведенном примере на MS SQL Server возвращала одно-единственное значение в качестве результата. При попытке организовать аналогичным образом работу с процедурой, возвращающей многострочный набор данных, результатом будет все равно одна единственная строка. Однако это не означает, что в СУБД, не поддерживающих селективные ХП, с помощью процедуры нельзя получить многострочные результаты.

Рассмотрим, как можно реализовать в MS SQL Server и Oracle процедуру выбора ListAbonent, которая в подпараграфе 6.2.3 создавалась как селективная для Firebird и возвращала несколько строк в качестве результата.

Самый простой способ реализации процедур выбора в MS SQL Server — выборка данных запросом SELECT в теле процедуры и по-

следующий вызов процедуры для вывода информации. Создадим процедуру в MS SQL Server следующим образом:

```
CREATE PROCEDURE ListAbonent
    @Mes INT, @God INT
AS
    SET NOCOUNT ON;
    SELECT A.AccountCD, A.Fio, P.PayDate,
           P.PaySum, P.PayMonth, P.PayYear
    FROM Abonent A, PaySumma P
    WHERE A.AccountCD = P.AccountCD AND P.PaySum > 500 AND
           P.PayMonth = @Mes AND P.PayYear = @God;
GO
```

В данном случае процедура создается без выходных параметров. Получить результат выполнения можно путем простого вызова процедуры:

```
EXEC ListAbonent 10, 2018;
```

Результат выполнения будет совпадать с результатом, приведенным ранее на рис. 6.6 (за исключением формата вывода дат).

Еще один распространенный и часто применимый на практике метод — получение набора данных, возвращаемого процедурой выбора, через временную таблицу. Для получения результатов необходимо:

- 1) создать временную таблицу;
- 2) вставить в нее результаты выполнения процедуры ListAbonent, используя запрос INSERT... EXEC...;
- 3) выбрать данные из этой таблицы;
- 4) удалить таблицу.

Код будет выглядеть так:

```
CREATE TABLE #tempTable
    (AccountCD NVARCHAR(6), Fio NVARCHAR(20), PayDate DATE,
     PaySum MONEY, PayMonth INT, PayYear INT)
INSERT INTO #tempTable (AccountCD, Fio, PayDate,
                        PaySum, PayMonth, PayYear)
    EXEC ListAbonent 10, 2018
SELECT * FROM #tempTable
DROP TABLE #tempTable
```

Процедуру выбора ListAbonent в MS SQL Server также можно реализовать с использованием типа CURSOR в качестве типа выходного результата. В таком случае скрипт создания процедуры должен быть следующий:

```
CREATE PROCEDURE ListAbonent
    @Mes INT, @God INT, @CurrencyCursor CURSOR VARYING OUTPUT
```

```

AS
    SET NOCOUNT ON;
    SET @CurrencyCursor = CURSOR
    FORWARD_ONLY STATIC FOR
        SELECT A.AccountCD, A.Fio, P.PayDate,
            P.PaySum, P.PayMonth, P.PayYear
        FROM Abonent A, PaySumma P
        WHERE A.AccountCD = P.AccountCD AND P.PaySum > 500 AND
            P.PayMonth = @Mes AND P.PayYear = @God;
    OPEN @CurrencyCursor;
GO

```

В приведенном скрипте для процедуры объявляется выходной параметр @CurrencyCursor типа CURSOR. С помощью оператора SET параметру устанавливается в соответствие курсор, определяемый для запроса SELECT. Опция FORWARD_ONLY указывает, что курсор может просматриваться только от первой записи к последней, опция STATIC позволяет создать временную копию данных для использования курсором. В конце создания процедуры выполняется открытие курсора оператором OPEN.

Для получения результатов необходимо выполнить пакет:

```

DECLARE @MyCursor CURSOR;
EXEC ListAbonent @Mes = 10, @God = 2018,
    @CurrencyCursor = @MyCursor OUTPUT;
WHILE (@@FETCH_STATUS = 0)
BEGIN;
    FETCH NEXT FROM @MyCursor;
END;
CLOSE @MyCursor;
DEALLOCATE @MyCursor;
GO

```

Пакет объявляет локальную переменную @MyCursor, после чего выполняет процедуру ListAbonent, присваивающую курсор @CurrencyCursor локальной переменной. В данном случае оператор EXEC использует именованное присвоение параметров. Затем в цикле выбираются строки из курсора, после чего он закрывается и уничтожается.

Результат выполнения совпадает с результатом на рис. 6.6.

Дополнительные примеры различных ХП MS SQL Server (с параметрами-шаблонами, с возвратом табличных значений параметров, с обработкой ошибок и т. д.) приведены в работе [7].

В СУБД Oracle реализовать процедуру выбора ListAbonent можно, как и в MS SQL Server, с использованием временной таблицы. При этом в Oracle временная таблица должна быть создана до создания процедуры, и вставка данных в нее должна выполняться внутри процедуры запросом INSERT... SELECT... . Скрипт создания таблицы и процедуры будет следующим:


```

CREATE GLOBAL TEMPORARY TABLE tempTable
  (AccountCD VARCHAR2(6), Fio VARCHAR2(30), PayDate DATE,
   PaySum NUMERIC(15,2), PayMonth SMALLINT, PayYear SMALLINT);
CREATE PROCEDURE ListAbonent (Mes INTEGER, God INTEGER)
AS
BEGIN
  INSERT INTO tempTable
  SELECT A.AccountCD, A.Fio, P.PayDate,
         P.PaySum, P.PayMonth, P.PayYear
  FROM Abonent A, PaySumma P
  WHERE A.AccountCD = P.AccountCD AND P.PaySum > 500 AND
        P.PayMonth = Mes AND P.PayYear = God;

END;
/

```

Получить результаты выполнения процедуры (см. рис. 6.6) в Oracle можно с помощью вызова процедуры и затем выборки данных из временной таблицы:

```

EXEC ListAbonent (10,2018);
SELECT * FROM tempTable;

```

Следует заметить, что временная таблица была создана по умолчанию как ON COMMIT DELETE ROWS, поэтому данные в ней существуют в пределах одной транзакции. Если, например, попытаться выполнить еще раз выборку данных из таблицы, не вызывая перед этим процедуру ListAbonent, то будет получен пустой результат.

Реализовать предыдущую ХП в Oracle можно и проще — с использованием пакета DBMS_OUTPUT. Скрипт создания процедуры выбора будет таким:

```

CREATE PROCEDURE ListAbonent (Mes INTEGER, God INTEGER)
AS
BEGIN
  FOR item IN
    (SELECT A.AccountCD, A.Fio, P.PayDate,
             P.PaySum, P.PayMonth, P.PayYear
     FROM Abonent A, PaySumma P
     WHERE A.AccountCD = P.AccountCD AND P.PaySum > 500 AND
           P.PayMonth = Mes AND P.PayYear = God)
  LOOP
    DBMS_OUTPUT.PUT_LINE(item.AccountCD || ' ' || item.Fio || ' '
|| item.PayDate || ' ' || item.PaySum || ' ' || item.PayMonth || ' '
|| item.PayYear);
  END LOOP;
END;
/

```

В приведенном скрипте организуется цикл FOR для последовательного перебора возвращаемых запросом строк. Для каждой такой

строки вызывается метод PUT_LINE пакета DBMS_OUTPUT, который выводит данные на экран. После вызова процедуры результаты выполнения будут видны в окне вывода dbForge Studio (рис. 6.9).

015527	Конюхов В. С.	03.11.18	611,3	10	2018
136169	Денисова Е. К.	26.11.18	528,44	10	2018

Рис. 6.9. Результат использования пакета DBMS_OUTPUT

Процедуры, возвращающие многострочный результат, в Oracle можно реализовывать также с использованием курсора и, кроме того, получать наборы данных в Oracle можно с помощью pipelined-функций. Примеры процедур с курсорами будут рассмотрены далее в этом подпараграфе, а пример pipelined-функции будет приведен в параграфе 6.3.

Рассмотрим следующий пример SQL-сценария, показывающий возможности работы с выполняемыми процедурами с использованием рекурсии. В примере выполняется расчет факториалов, для чего создаются таблица Ftable и две выполняемые процедуры с именами Factorial [19, 20] и FactorialSet.

В Firebird скрипт создания таблицы и процедур будет выглядеть так:

```
CREATE TABLE Ftable (Fnum INTEGER, Fvalue DOUBLE PRECISION);
SET TERM !! ;
CREATE PROCEDURE Factorial (Num INTEGER)
RETURNS (N_Factorial DOUBLE PRECISION)
AS
BEGIN
    IF (Num < 0) THEN EXIT;
    IF (Num = 1 OR Num = 0) THEN
        BEGIN
            N_Factorial = 1;
            EXIT;
        END
    ELSE
        BEGIN
            EXECUTE PROCEDURE Factorial (Num-1)
            RETURNING_VALUES N_Factorial;
            N_Factorial = N_Factorial * Num;
            EXIT;
            WHEN ANY DO
                N_Factorial = NULL;
        END
    END
END !!
CREATE PROCEDURE FactorialSet (Minf INTEGER, Maxf INTEGER)
AS
    DECLARE VARIABLE i INTEGER;
    DECLARE VARIABLE f DOUBLE PRECISION;
```

```

BEGIN
    DELETE FROM Ftable;
    IF (Maxf < Minf) THEN EXIT;
    i = Minf;
    WHILE ( i <= Maxf) DO
    BEGIN
        EXECUTE PROCEDURE Factorial (:i) RETURNING_VALUES f;
        IF (:f IS NOT NULL) THEN INSERT INTO Ftable (Fnum, Fvalue)
            VALUES (:i, :f);

        i = i + 1;
    END
END !!
SET TERM ; !!

```

Таблица Ftable предназначена для хранения значений факториала. В столбце Fnum хранится число, для которого вычисляется факториал, а в столбце Fvalue — само значение факториала.

Для типа DOUBLE PRECISION (тип выходного параметра) возможно вычисление факториала (без переполнения разрядной сетки) для значения 170, поэтому введена дополнительная проверка входного значения.

Выполняемая процедура Factorial вычисляет значение факториала рекурсивным методом. Если на входе процедуры ноль или единица, то значение факториала (N_Factorial) принимается равным единице и оператор EXIT осуществляет возврат выходного параметра и выход из процедуры. Если на входе целое положительное число, отличное от нуля или единицы, то происходит рекурсивное вычисление факториала с использованием соотношения

$$\text{Num!} = \text{Num} * (\text{Num} - 1)!$$

В результате обработки исключительной ситуации в случае переполнения разрядной сетки процедура Factorial возвращает NULL.

Рекурсивный вызов процедуры в Firebird осуществляется при вызове процедурой самой себя запросом EXECUTE PROCEDURE.

Примечание. При рекурсивном вызове в Firebird допускается глубина вложенности до 1000 уровней, однако нужно помнить о том, что свободные ресурсы на сервере могут закончиться раньше, чем будет достигнута максимальная вложенность ХП [21].

Оператор вызова процедуры EXECUTE PROCEDURE может также использоваться и в SQL-редакторе IBExpert. Например, чтобы вызвать процедуру Factorial в интерактивном режиме и передать ей в качестве параметра значение 170, необходимо выполнить запрос

```
EXECUTE PROCEDURE Factorial (170);
```

Результат вызова процедуры Factorial представлен на рис. 6.10, а.

-----Procedure executing results:-----
N_FACTORIAL = 7,25741561530799E306

N_FACTORIAL

7,25741561530799E306

а

б

Рис. 6.10. Результат вызова процедуры Factorial

Если в теле процедуры Factorial команду EXIT заменить командой SUSPEND, то получится селективная процедура (результат ее вызова — на рис. 6.10, б):

```
SELECT * FROM Factorial (170);
```

Удалить процедуру Factorial нельзя из-за того, что ее использует процедура FactorialSet.

Выполняемая процедура FactorialSet предназначена для вычисления значений факториала чисел из некоторого диапазона и занесения полученных результатов в таблицу Ftable. Процедура получает в качестве параметров первое (Minf) и последнее (Maxf) значения диапазона вычисления факториала. В теле процедуры осуществляется цикл от Minf до Maxf с шагом, равным единице. Для каждого значения параметра цикла (локальная переменная i) вычисляется значение факториала путем вызова процедуры Factorial. Число, для которого вычислен факториал, и значение факториала заносятся в таблицу Ftable. Например, чтобы вычислить пять значений факториала от 1 до 5, необходимо в интерактивном режиме выполнить запрос

```
EXECUTE PROCEDURE FactorialSet (1, 5);
```

Для просмотра результатов нужно выполнить запрос для таблицы Ftable:

```
SELECT * FROM Ftable;
```

Результат выполнения запроса представлен на рис. 6.11.

FNUM	FVALUE
1	1,000
2	2,000
3	6,000
4	24,000
5	120,000

Рис. 6.11. Таблица Ftable со значениями факториалов от 1 до 5

Рассмотрим, как можно решить аналогичную задачу в MS SQL Server. Для создания таблицы Ftable и процедур Factorial и FactorialSet в MS SQL Server выполним следующий скрипт:

```
CREATE TABLE Ftable (Fnum INTEGER, Fvalue DOUBLE PRECISION);
GO
CREATE PROCEDURE Factorial
  (@Num INTEGER,@N_Factorial DOUBLE PRECISION OUTPUT)
AS
  DECLARE @Num_Less_One INTEGER;
  DECLARE @vN_Factorial INTEGER;
BEGIN
  IF (@Num < 0 OR @Num > 170) BEGIN RETURN(1) END;
  IF (@Num = 1 OR @Num = 0) BEGIN
    SET @N_Factorial = 1;
  END
  ELSE BEGIN
    SET @Num_Less_One = @Num - 1;
    EXECUTE Factorial @Num_Less_One, @vN_Factorial OUTPUT;
    SET @N_Factorial = @vN_Factorial * @Num;
  END
END
GO
CREATE PROCEDURE FactorialSet (@Minf INTEGER, @Maxf INTEGER)
AS
  DECLARE @i INTEGER;
  DECLARE @f DOUBLE PRECISION;
BEGIN
  DELETE FROM Ftable;
  IF (@Maxf < @Minf) RETURN;
  SET @i = @Minf;
  WHILE ( @i <= @Maxf)
  BEGIN
    EXEC Factorial @i, @f OUTPUT;
    INSERT INTO Ftable (Fnum, Fvalue) VALUES (@i, @f);
    SET @i = @i + 1;
  END
END
GO
```

Для вычисления с помощью созданных процедур пяти значений факториала от 1 до 5 и просмотра результатов необходимо выполнить запросы:

```
EXEC FactorialSet 1, 5;
SELECT * FROM Ftable;
```

Результаты будут аналогичны приведенным на рис. 6.11 для Firebird.

Если же попытаться, например, выполнить запрос

```
DECLARE @N_Factorial DOUBLE PRECISION;
EXEC Factorial 170,@N_Factorial OUTPUT;
```

то получится не корректный результат, как в Firebird, а ошибка «Превышен максимальный уровень вложенности хранимой процедуры, функции, триггера или представления (limit 32)».

Примечание. При рекурсивном вызове в MS SQL Server допускается глубина вложенности не более 32 уровней.

Таким образом, в MS SQL Server с помощью созданной процедуры Factorial можно вычислить только значение факториала числа не более 32.

Рассмотрим особенности создания и использования вложенных рекурсивных процедур в СУБД Oracle, а также отличия при работе с входными и выходными процедурными параметрами. Скрипт создания таблицы и процедур будет таким:

```
CREATE TABLE Ftable (Fnum INTEGER, Fvalue DOUBLE PRECISION);
CREATE PROCEDURE Factorial
  (Num IN INTEGER, N_Factorial OUT DOUBLE PRECISION)
IS
  Num_Less_One INTEGER;
  vN_Factorial INTEGER;
BEGIN
  IF (Num < 0 OR Num > 170) THEN
    RETURN;
  END IF;
  IF (Num = 1 OR Num = 0) THEN
    N_Factorial := 1;
  ELSE
    Num_Less_One := Num - 1;
    Factorial (Num_Less_One, vN_Factorial);
    N_Factorial := vN_Factorial * Num;
  END IF;
END Factorial;
/
CREATE PROCEDURE FactorialSet (Minf INTEGER, Maxf INTEGER)
IS
  i INTEGER;
  f DOUBLE PRECISION;
BEGIN
  DELETE FROM Ftable;
  IF (Maxf < Minf) THEN
    RETURN;
  END IF;
  i := Minf;
  WHILE ( i <= Maxf) LOOP
    Factorial (i, f);
    INSERT INTO Ftable (Fnum, Fvalue) VALUES (i, f);
    i := i + 1;
  END LOOP;
END FactorialSet;
/
```

Процедура Factorial создается с входным (IN) параметром Num и выходным (OUT) параметром N_Factorial. В Oracle значение параметра вида IN передается в процедуру при ее вызове, внутри процедуры входной параметр рассматривается в качестве параметра только для чтения — он не может быть изменен. Параметр вида OUT внутри процедуры рассматривается как параметр только для записи — ему можно присвоить значение, но считать из него значение нельзя. Так, нельзя до вызова процедуры Factorial присвоить переменной vN_Factorial значение и затем при вызове передать это значение выходному параметру N_Factorial с целью использования его для каких-либо вычислений. Когда процедура завершается и управление передается в вызывающую среду, входной параметр не изменяется, а содержимое выходного параметра присваивается переменной вызывающей среды.

При рекурсивном вызове самой себя процедурой Factorial и при вызове из процедуры FactorialSet оператор вызова EXEC опущен, вызов осуществляется указанием только имени процедуры и ее параметров. При попытке использовать оператор вызова возникнет ошибка.

Следует обратить внимание на то, что операторы создания процедур в скрипте Oracle разделены между собой косой чертой, «/».

Вычислить и увидеть результаты (рис. 6.11) факториала от 1 до 5 можно с помощью запросов

```
EXEC FactorialSet (1, 5);  
SELECT * FROM Ftable;
```

Oracle, в отличие от MS SQL Server, никак не ограничивает глубину вложенности процедур при рекурсивном вызове. Ответственность за возможное возникновение закливания и исчерпание оперативной памяти СУБД перекладывает на программистов. Максимальное число для вычисления факториала в Oracle ограничивается лишь точностью типа для представления результата DOUBLE PRECISION. При этом можно вычислить только значение факториала числа не более 34, хотя тот же тип DOUBLE PRECISION в Firebird, например, позволяет вычислить факториал числа включительно до 170. При попытке вычислить факториал числа более 34 в Oracle возникает ошибка «ORA-06502: PL/SQL: numeric or value error: number precision too large».

Это связано с тем, что стандарт SQL не определяет жестко тип DOUBLE PRECISION, а требует только, чтобы точность числа типа DOUBLE PRECISION была больше, чем точность типа REAL. Поэтому точность типа DOUBLE PRECISION зависит от реализации в конкретной СУБД. Любое число можно представить в виде мантиссы, умноженной на число 10, возведенное в степень, показатель которой яв-

ляется экспонентой. В некоторых СУБД тип имеет и для мантиссы, и для экспоненты в два раза большую вместимость, чем тип REAL. В других системах тип DOUBLE PRECISION дает больше, чем двойная вместимость мантиссы, и меньше, чем двойная вместимость типа REAL для экспоненты. В некоторых системах тип DOUBLE PRECISION дает для мантиссы вместимость в два раза больше, чем у типа REAL, а для экспоненты — ту же, что и у REAL (точность удваивается, а диапазон — нет). Если сравнить результаты вычисления факториала числа 34, например, в Firebird и Oracle, то можно заметить, что в Firebird большое число представлено в виде приблизительного числового типа (2,95232799039604E38), а в Oracle это точное число без представления в виде мантиссы и экспоненты.

Рассмотрим пример еще одного SQL-сценария, в котором создаются таблица Saldo и выполняемая процедура с именем ResultSaldo. Процедура ResultSaldo позволяет для заданного абонента по заданной услуге подсчитать остаток на конец каждого года заданного периода. Скрипт в Firebird:

```
CREATE TABLE Saldo (  
    Account VARCHAR(6) NOT NULL REFERENCES Abonent ON UPDATE CASCADE,  
    Service PKFIELD NOT NULL REFERENCES Services ON UPDATE CASCADE,  
    God Tyear,  
    Begin_Ostatok MONEY,  
    Nachisl MONEY,  
    Pay MONEY,  
    Result_Ostatok MONEY  
);  
  
SET TERM !!;  
  
CREATE PROCEDURE ResultSaldo (  
    Account VARCHAR(6),  
    Serv PKFIELD,  
    Start_God Tyear,  
    Final_God Tyear,  
    Ostat MONEY  
)  
RETURNS (Result_Ostat MONEY)  
AS  
DECLARE VARIABLE Begin_Ostatok MONEY = 0;  
DECLARE VARIABLE Nach MONEY = 0;  
DECLARE VARIABLE Pay MONEY = 0;  
DECLARE VARIABLE PredBegin_Ostatok MONEY = 0;  
DECLARE VARIABLE Pred_Nachisl MONEY = 0;  
DECLARE VARIABLE Pred_Pay MONEY = 0;  
DECLARE VARIABLE PredResult_Ostatok MONEY = 0;  
  
BEGIN  
    IF (Start_God > Final_God) THEN  
        BEGIN
```



```

    Result_Ostat = Ostat;
    EXIT;
END

-- получение сумм начислений и оплат для абонента за заданный год
SELECT COALESCE((
    SELECT SUM(N.NachislSum) SumNach
    FROM NachislSumma N
    WHERE N.AccountCD = :Account
        AND N.ServiceCD = :Serv
        AND N.NachislYear = :Start_God
    GROUP BY N.AccountCD, N.ServiceCD, N.NachislYear
), 0),
COALESCE((
    SELECT SUM(P.PaySum) SumPay
    FROM PaySumma P
    WHERE P.AccountCD = :Account
        AND P.ServiceCD = :Serv
        AND P.PayYear = :Start_God
    GROUP BY P.AccountCD, P.ServiceCD, P.PayYear
), 0)
FROM RDB$DATABASE
INTO :Nach, :Pay;

-- изменение остатка
Begin_Ostatok = Ostat;

Ostat = Ostat + (Nach - Pay);

-- запись результата в таблицу Saldo
SELECT Begin_Ostatok, Nachisl, Pay, Result_Ostatok
FROM Saldo
WHERE Account = :Account
    AND Service = :Serv
    AND God = :Start_God
INTO :PredBegin_Ostatok, :Pred_Nachisl,
    :Pred_Pay, :PredResult_Ostatok;

IF (ROW_COUNT = 0) THEN
    INSERT INTO Saldo
    VALUES (:Account, :Serv, :Start_God, :Begin_Ostatok,
        :Nach, :Pay, :Ostat
    );
ELSE IF (
    PredBegin_Ostatok <> Begin_Ostatok
    OR Pred_Nachisl <> Nach
    OR Pred_Pay <> Pay
    OR PredResult_Ostatok <> Ostat
) THEN
    UPDATE Saldo
    SET Begin_Ostatok = :Begin_Ostatok,
        Nachisl = :Nach,
        Pay = :Pay,

```

```

        Result_Ostatok = :Ostat
WHERE Account = :Account
      AND Service = :Serv
      AND God = :Start_God;

-- переход к следующему году
Start_God = Start_God + 1;

-- рекурсивный вызов процедуры для следующего года периода
EXECUTE PROCEDURE ResultSaldo(:Account, :Serv, :Start_God,
    :Final_God, :Ostat) RETURNING_VALUES Result_Ostat;
END!!

SET TERM; !!

```

В данном скрипте сначала создается таблица Saldo для хранения значений денежных остатков абонентов. В столбце Account этой таблицы хранится номер лицевого счета абонента, в столбце Service — код услуги, в столбце God — год, за который рассчитывается остаток, в столбце Begin_Ostatok — значение остатка на начало года, в столбце Nachisl — сумма, начисленная за год, в столбце Pay — сумма, оплаченная за год, а в столбце Result_Ostatok — вычисленное значение остатка на конец года.

Выполняемая процедура ResultSaldo рекурсивным методом вычисляет для определенного абонента (Account) по указанной услуге (Serv) значение денежного остатка как разницу между суммами значений начислений и оплат за период, заданный начальным (Start_God) и конечным (Final_God) годами. Значение начального остатка (Ostat) задается при вызове процедуры, а затем в теле процедуры его значение изменяется после извлечения запросом данных о начислении и оплате за очередной год периода. После этого анализируются данные таблицы Saldo. Если в таблице Saldo нет данных о денежных остатках заданного абонента по указанной услуге за определенный год, то выполняется вставка новой строки в таблицу. Если такие данные есть, но хранимый в таблице начальный остаток (Begin_Ostatok), сумма начислений (Nachisl), сумма оплат (Pay) или конечный остаток (Result_Ostatok) отличаются от вычисленных процедурой (соответственно Begin_Ostatok, Nach, Pay и Ostat), то происходит обновление значений в таблице. Если же остатки совпадают, то модификация данных таблицы не выполняется. После выполнения описанных действий с данными таблицы Saldo происходит переход к следующему году периода.

Рекурсивное выполнение процедуры осуществляется при вызове процедурой самой себя запросом EXECUTE PROCEDURE. При каждом рекурсивном вызове на вход поступают очередной год расчета (Start_God), а также значение остатка, рассчитанное к моменту вызова (Ostat).

Выполнение процедуры продолжается до тех пор, пока год, за который выполняется расчет (Start_God), не станет больше года окончания заданного периода (Final_God). В таком случае результирующее значение остатка за период (Result_Ostat) принимается равным входному начальному остатку (Ostat), и оператор EXIT осуществляет возврат выходного параметра и выход из процедуры.

Выполнить процедуру ResultSaldo можно в SQL-редакторе IVExpert. Например, для вычисления денежного остатка абонента с номером лицевого счета '015527' по услуге с кодом, равным 1, за период с 2017 по 2018 г. при заданном нулевом начальном остатке можно выполнить запрос

```
EXECUTE PROCEDURE ResultSaldo ('015527', 1, 2017, 2018, 0);
```

Результат вызова процедуры ResultSaldo представлен на рис. 6.12.

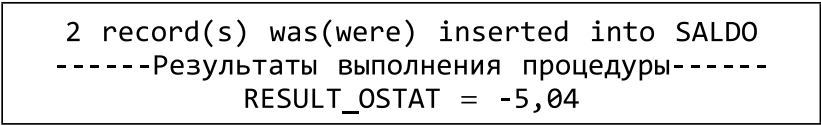


Рис. 6.12. Результат вызова процедуры ResultSaldo

Таким образом, на рисунке представлен результирующий остаток абонента по услуге с кодом 1 по 2018 год. Расшифровка полученного результата по годам периода в ходе выполнения процедуры добавлена в таблицу Saldo. Для просмотра строк, добавленных в таблицу Saldo, необходимо выполнить запрос

```
SELECT * FROM Saldo ORDER BY God;
```

Результат выполнения запроса представлен на рис. 6.13.

ACCOUNT	SERVICE	GOD	BEGIN_OSTATOK	NACHISL	PAY	RESULT_OSTATOK
015527	1	2017	0	46,64	50	-3,36
015527	1	2018	-3,36	38,32	40	-5,04

Рис. 6.13. Таблица Saldo со значениями остатков по годам

В каждой строке данной таблицы хранится значение остатка, полученного как сумма остатков за указанный год и за предшествующий этому году период. Если сравнить полученный результат с результатом выполнения описанной ранее селективной процедуры Oborot_Saldo_Vedomost (см. рис. 6.7), то можно убедиться, что остатки на конец 2017 г. и на конец 2018 г. совпадают с остатками соответственно на конец декабря 2017 г. и на конец декабря 2018 г., которые были получены с помощью процедуры Oborot_Saldo_Vedomost.

Реализуем задачу по вычислению денежных остатков абонентов в MS SQL Server. Скрипт создания таблицы Saldo и процедуры ResultSaldo будет таким:

```
CREATE TABLE Saldo
(Account VARCHAR(6) NOT NULL REFERENCES Abonent ON UPDATE CASCADE,
Service PKFIELD NOT NULL REFERENCES Services ON UPDATE CASCADE,
God TYear, Begin_Ostatok Money, Nachisl Money,
Pay Money, Result_Ostatok Money);
GO
CREATE PROCEDURE ResultSaldo
(@Account VARCHAR(6), @Serv PKFIELD, @Start_God Tyear,
@Final_God Tyear, @Ostat Money, @Result_Ostat MONEY OUTPUT)
AS
BEGIN
    DECLARE @Begin_Ostatok Money = 0;
    DECLARE @Nach Money = 0;
    DECLARE @Pay Money = 0;
    DECLARE @PredBegin_Ostatok Money = 0;
    DECLARE @Pred_Nachisl Money = 0;
    DECLARE @Pred_Pay Money = 0;
    DECLARE @PredResult_Ostatok Money = 0;
    IF (@Start_God > @Final_God)
    BEGIN
        SET @Result_Ostat = @Ostat;
        RETURN;
    END

    -- получение сумм начислений и оплат для абонента
    -- за заданный год
    SELECT
        @Nach = COALESCE ((SELECT SUM(N.NachislSum) SumNach
                           FROM NachislSumma N
                           WHERE N.AccountCD = @Account
                                AND N.ServiceCD = @Serv
                                AND N.NachislYear = @Start_God
                           GROUP BY N.AccountCD, N.ServiceCD,
                                   N.NachislYear), 0),
        @Pay = COALESCE ((SELECT SUM(P.PaySum) SumPay
                           FROM PaySumma P
                           WHERE P.AccountCD = @Account
                                AND P.ServiceCD = @Serv
                                AND P.PayYear = @Start_God
                           GROUP BY P.AccountCD, P.ServiceCD,
                                   P.PayYear), 0)

    -- изменение остатка
    SET @Begin_Ostatok = @Ostat;
    SET @Ostat = @Ostat + (@Nach - @Pay);

    -- запись результата в таблицу Saldo
    SELECT @PredBegin_Ostatok = Begin_Ostatok,
           @Pred_Nachisl = Nachisl,
```

```

        @Pred_Pay = Pay,
        @PredResult_Ostatok = Result_Ostatok
FROM Saldo
WHERE Account = @Account AND Service = @Serv
        AND God = @Start_God;

IF (@@ROWCOUNT = 0)
    INSERT INTO Saldo
    VALUES (@Account, @Serv, @Start_God,
            @Begin_Ostatok, @Nach, @Pay, @Ostat);
ELSE
    IF (@PredBegin_Ostatok <> @Begin_Ostatok
        OR @Pred_Nachisl <> @Nach
        OR @Pred_Pay <> @Pay
        OR @PredResult_Ostatok <> @Ostat)
        UPDATE Saldo SET Begin_Ostatok = @Begin_Ostatok,
            Nachisl = @Nach,
            Pay = @Pay,
            Result_Ostatok = @Ostat
    WHERE Account = @Account
        AND Service = @Serv
        AND God = @Start_God;

-- переход к следующему году
SET @Start_God = @Start_God + 1;
-- рекурсивный вызов процедуры для следующего года периода
EXEC ResultSaldo @Account, @Serv, @Start_God,
    @Final_God, @Ostat, @Result_Ostat OUTPUT;
END;
GO

```

Выполнить процедуру ResultSaldo в MS SQL Server можно, например, запросом

```

DECLARE @vResult_Ostat MONEY;
EXEC ResultSaldo '015527', 1, 2017, 2018, 0, @vResult_Ostat OUTPUT;
SELECT @vResult_Ostat;

```

Результатом работы процедуры будет вывод итогового остатка абонента (аналогично рис. 6.12) и набор строк в таблице Saldo (рис. 6.13).

Решим аналогичную задачу в Oracle, продемонстрировав при этом использование курсора и параметров вида IN OUT в ХП Oracle. Скрипт будет выглядеть так:

```

CREATE TABLE Saldo (
    Account VARCHAR2(6) NOT NULL REFERENCES Abonent,
    Service INTEGER NOT NULL REFERENCES Services,
    God SMALLINT,
    Begin_Ostatok NUMERIC(15, 2),
    Nachisl NUMERIC(15, 2),
    Pay NUMERIC(15, 2),
    Result_Ostatok NUMERIC(15, 2)
);

```

```

CREATE PROCEDURE ResultSaldo (
    Account VARCHAR2,
    Serv INTEGER,
    Start_God IN OUT SMALLINT,
    Final_God SMALLINT,
    Ostat IN OUT REAL
) IS vBegin_Ostatok NUMERIC(15, 2) := 0;
    vNach NUMERIC(15, 2) := 0;
    vPay NUMERIC(15, 2) := 0;
    CURSOR Get_Saldo IS
        SELECT Begin_Ostatok, Nachisl, Pay, Result_Ostatok
        FROM Saldo
        WHERE Account = Account
            AND Service = Serv
            AND God = Start_God;
    vGetSaldo Get_Saldo % ROWTYPE;

BEGIN
    IF (Start_God > Final_God) THEN
        RETURN;
    END IF;

    -- получение сумм начислений и оплат для абонента
    -- за заданный год
    SELECT COALESCE((
        SELECT SUM(N.NachislSum) SumNach
        FROM NachislSumma N
        WHERE N.AccountCD = Account
            AND N.ServiceCD = Serv
            AND N.NachislYear = Start_God
        GROUP BY N.AccountCD,
            N.ServiceCD,
            N.NachislYear
        ), 0),
    COALESCE((
        SELECT SUM(P.PaySum) SumPay
        FROM PaySumma P
        WHERE P.AccountCD = Account
            AND P.ServiceCD = Serv
            AND P.PayYear = Start_God
        GROUP BY P.AccountCD,
            P.ServiceCD,
            P.PayYear
        ), 0)
    INTO vNach, vPay
    FROM DUAL;

    -- изменение остатка
    vBegin_Ostatok := Ostat;
    Ostat := Ostat + (vNach - vPay);

    -- запись результата в таблицу Saldo
    OPEN Get_Saldo;

```

```

FETCH Get_Saldo
INTO vGetSaldo;
IF (Get_Saldo % ROWCOUNT = 0) THEN
    INSERT INTO Saldo
    VALUES (Account, Serv, Start_God, vBegin_Ostatok,
            vNach, vPay, Ostat);
ELSE IF (
    vGetSaldo.Begin_Ostatok <> vBegin_Ostatok
    OR vGetSaldo.Nachisl <> vNach
    OR vGetSaldo.Pay <> vPay
    OR vGetSaldo.Result_Ostatok <> Ostat
) THEN
    UPDATE Saldo
    SET Begin_Ostatok = vBegin_Ostatok,
        Nachisl = vNach,
        Pay = vPay,
        Result_Ostatok = Ostat
    WHERE Account = Account
        AND Service = Serv
        AND God = Start_God;
END IF;
END IF;
CLOSE Get_Saldo;

-- переход к следующему году
Start_God := Start_God + 1;

-- рекурсивный вызов процедуры для следующего года периода
ResultSaldo(Account, Serv, Start_God, Final_God, Ostat);
END;
/

```

Процедура ResultSaldo создается с входными параметрами Account, Serv, Final_God и параметрами Start_God и Ostat комбинированного вида IN OUT, которые используются для одновременного приема и возврата значений. Следует обратить внимание, что в Oracle для параметров ХП указывается только тип данных, а длина и точность не указываются. Так, длина не указана для входного параметра Account и комбинированного параметра Ostat. Oracle самостоятельно определяет длину и точность для параметров из контекста, в связи с чем к заданию типов для параметров ХП следует отнестись внимательно. Например, параметр процедуры Ostat объявлен с типом REAL, а не NUMERIC, хотя целевая таблица Saldo хранит остатки в типе NUMERIC(15,2). Это связано с тем, что для типа NUMERIC по умолчанию точность равна 0, и, если его использовать для параметра процедуры, то на выходе будем получать округленные до целого значения. Точность типа REAL по умолчанию ненулевая, поэтому результат будет корректен.

Получение сумм начислений (Nach) и оплат (Pay) для абонента за заданный год выполняется запросом SELECT с секцией INTO,

которая указывается перед секцией FROM. Для работы с запросом, анализирующим данные таблицы Saldo, объявляются курсор Get_Saldo и переменная vGetSaldo курсорного типа (задается через Get_Saldo%ROWTYPE). Выборка из курсора выполняется вне цикла, так как заведомо он возвращает либо 0, либо 1 строку. Количество возвращаемых строк анализируется с использованием курсорного атрибута %ROWCOUNT. Если бы запрос не был оформлен как курсор, а определялся бы непосредственно в теле ХП, то количество возвращаемых им строк можно было бы получить с помощью переменной SQL%ROWCOUNT.

После модификации таблицы Saldo выполняется переход к следующему году путем изменения параметра Start_God. Следует отметить, что если бы параметр Start_God был объявлен не как IN OUT, а просто как входной параметр, попытка его изменения в коде процедуры вызвала бы ошибку.

Выполнить созданную процедуру в Oracle можно через анонимный PL/SQL блок (см. параграф 6.5) вида

```
DECLARE
    vGod SMALLINT;
    vOstat SMALLINT;
BEGIN
    vGod := 2017;
    vOstat := 0;
    ResultSaldo ('015527', 1, vGod, 2018, vOstat);
END;
```

Рассмотрим, в каких еще случаях может оказаться полезным использование ХП. В подпараграфе 5.3.2 отмечалось, что не рекомендуется применение целевой таблицы в запросе DELETE с подзапросом. Например, для Firebird был приведен запрос DELETE, удаляющий информацию обо всех оплатах тех абонентов, которыми было сделано более восьми платежей. Такая задача может быть довольно просто решена с помощью выполняемой ХП. Все приведенные процедуры используют курсоры в том или ином виде.

В Firebird скрипт создания процедуры будет таким:

```
CREATE PROCEDURE Del
AS
    DECLARE VARIABLE Fld VARCHAR(6);
BEGIN
    FOR SELECT AccountCd
        FROM PaySumma
        GROUP BY AccountCd
        HAVING COUNT(PayFactCd) > 8
    INTO :Fld
    DO
        DELETE FROM PaySumma WHERE AccountCD = :Fld;
END;
```


Скрипт для создания аналогичной процедуры в MS SQL Server:

```
CREATE PROCEDURE Del
AS
BEGIN
    DECLARE @Fld VARCHAR(6);
    DECLARE cursAcc CURSOR
        FOR SELECT AccountCd
            FROM PaySumma
            GROUP BY AccountCd
            HAVING COUNT(PayFactCd) > 8
    OPEN cursAcc;
    WHILE @@FETCH_STATUS = 0
    BEGIN
        FETCH NEXT FROM cursAcc INTO @Fld
        DELETE FROM PaySumma WHERE AccountCD = @Fld;
    END;
    CLOSE cursAcc;
    DEALLOCATE cursAcc;
END;
```

Скрипт для СУБД Oracle:

```
CREATE PROCEDURE Del
AS
    Fld VARCHAR2(6);
    CURSOR cursAcc IS
        SELECT AccountCd
        FROM PaySumma
        GROUP BY AccountCd
        HAVING COUNT(PayFactCd) > 8;
BEGIN
    FOR a IN cursAcc LOOP
        DELETE FROM PaySumma WHERE AccountCD = a.AccountCD;
    END LOOP;
END;
```

В хранимых процедурах можно использовать запросы модификации данных INSERT, UPDATE OR INSERT, UPDATE, DELETE и MERGE, содержащие специальную секцию для указания, что значения определенных столбцов в изменяемых строках должны запоминаться в соответствующих переменных, перечисленных в списке после INTO. В Firebird и Oracle специальная секция представляет собой секцию RETURNING, употребляемую в конце запросов модификации DML, а в MS SQL Server — секцию OUTPUT, употребляемую перед секцией VALUES (параграфы 5.1—5.4).

В Firebird и Oracle секция имеет формат

```
RETURNING <список_столбцов> INTO <список_переменных>;
```

где

```

<список_столбцов> ::= <возвращаемый_элемент1>
                        [[AS] псевдоним_элемента1]
                        [, <возвращаемый_элемент2> [[AS] псевдоним_элемента2] ]...

```

Столбцы, указанные в <список_столбцов>, могут быть любыми столбцами из целевой таблицы, в том числе обновляемыми.

Запросы модификации данных, использующие секцию RETURNING, должны оперировать не более чем одной строкой набора данных. Например, определение выполняемой ХП, использующей запрос на удаление данных с секцией RETURNING, в Firebird может выглядеть так:

```

CREATE PROCEDURE Save_Del (Req_CD INTEGER)
RETURNS (RAccountCD VARCHAR(6), RDate DATE)
AS
BEGIN
    DELETE FROM Request WHERE RequestCD = :Req_CD
    RETURNING AccountCD, IncomingDate
    INTO :RAccountCD, :RDate;
END;

```

Эта ХП удаляет информацию о ремонтной заявке с заданным номером (Req_CD). При этом номер лицевого счета абонента, подавшего удаляемую заявку, и дата подачи этой заявки выводятся как результат выполнения процедуры. Процедура может быть выполнена с помощью запроса

```
EXECUTE PROCEDURE Save_Del(1);
```

Результат выполнения процедуры представлен на рис. 6.14.

```

----- Результаты выполнения процедуры: -----
          RACCOUNTCD = 005488
          RDATE = 17.12.2016
1 записей было удалено из таблицы REQUEST

```

Рис. 6.14. Результат выполнения процедуры Save_Del

В Oracle скрипт создания процедуры будет таким:

```

CREATE PROCEDURE Save_Del
  (Req_CD IN INTEGER, RAccountCD OUT VARCHAR2, RDate OUT DATE)
AS
BEGIN
    DELETE FROM Request WHERE REQUESTCD = Req_CD
    RETURNING AccountCD, IncomingDate
    INTO RAccountCD, RDate;
END;
/

```

Чтобы выполнить процедуру, необходимо использовать анонимный PL/SQL блок (см. параграф 6.5):

```

DECLARE
    vAccountCD VARCHAR2(6);
    vDate DATE;
BEGIN
    Save_Del (1, vAccountCD, vDate);
    -- код использования переменных vAccountCD и vDate
    ...
END;
/

```

В СУБД MS SQL Server для сохранения столбцов строк, изменяемых запросами DML, используется секция OUTPUT, которая имеет формат

```

OUTPUT <список_сохраняемых_столбцов>
[INTO { @табличная_переменная | имя_таблицы } [(список_столбцов)] ]

```

где

```

<список_сохраняемых_столбцов> ::=
{ { DELETED | INSERTED | имя_таблицы } . { * | имя_столбца } |
$действие
| скалярное_выражение } [ [AS] псевдоним ] [ ,... ]

```

Секция OUTPUT используется, если нужно вывести или сохранить сразу весь набор строк, измененных запросом. Сохранение обычно выполняется либо в локальную переменную табличного типа TABLE, либо в таблицу БД.

При указании сохраняемых столбцов могут использоваться префиксы DELETED или INSERTED. Префикс столбца DELETED указывает на значение, удаляемое в результате выполнения операции обновления или удаления. Столбцы, имеющие префикс DELETED, отражают значение, которое было до завершения запроса UPDATE, DELETE или MERGE. Префикс DELETED не может указываться в секции OUTPUT запроса INSERT. Префикс столбца INSERTED указывает на значение, добавляемое в результате выполнении операции вставки или обновления. Столбцы, имеющие префикс INSERTED, отражают значение, полученное после завершения запроса UPDATE, INSERT или MERGE, но до выполнения триггеров. Префикс INSERTED не может указываться в секции OUTPUT запроса DELETE.

Помимо использования секции OUTPUT в DML-запросах, может применяться сохранение изменяемых строк в переменные в секции SET запроса UPDATE в виде

```

@<переменная> [= <столбец>] <операция> <выражение>

```

Процедуру по удалению заданной ремонтной заявки в MS SQL Server можно создать так:

```

CREATE PROCEDURE Save_Del

```

```

        (@Req_CD INTEGER)
AS
BEGIN
    DELETE FROM Request
    OUTPUT DELETED.*
    WHERE RequestCD = @Req_CD;
END;

```

В данной процедуре секция OUTPUT DELETED.* запроса DELETE возвращает все столбцы строк, удаляемых из таблицы Request.

Вызвать процедуру и просмотреть результаты (рис. 6.15) можно с помощью простого запроса

```
EXEC Save_Del 1;
```

В качестве заключительного примера приведем текст выполняемой ХП, реализующей добавление нового абонента с обработкой ошибок некорректного ввода оператора

```

CREATE EXCEPTION Excep '';
SET TERM !! ;
CREATE PROCEDURE Add_P(LAccoutntCD VARCHAR(20),LStreetCD Pkfield)
AS
BEGIN
    INSERT INTO Abonent(AccountCD, StreetCD) VALUES
    (:LAccoutntCD,:LStreetCD);
    WHEN
    SQLCODE -803 DO
    EXCEPTION Excep 'Дублирование строки';
    WHEN
    SQLSTATE '23000' DO
    EXCEPTION Ins_Restrict;
    END !!
    SET TERM ; !!

```

При возникновении ошибки добавления строки в таблицу Abonent производится замена системного текста ошибки пользовательским. Например, при попытке ввода абонента с дублирующимся номером лицевого счета

```
EXECUTE PROCEDURE Add_P ('005488', 1);
```

или улицей, отсутствующей в таблице Street,

```
EXECUTE PROCEDURE Add_P ('111111', 9);
```

выдается соответственно сообщение «Дублирование строки» или «Ограничение добавления строки»

Добавление абонента

```
EXECUTE PROCEDURE Add_P ('111111', 3);
```

происходит без ошибок.

REQUESTCD	ACCOUNT	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
1	005488	1	1	2016-12-17	2016-12-20	1

Рис. 6.15. Результат выполнения процедуры Save_Del в MS SQL Server

6.2.5. Динамический SQL

При создании приложений на процедурных языках часто возникает ситуация, когда структура SQL-запросов заранее не определена и может изменяться в зависимости от каких-либо условий. Для решения ситуаций подобного типа используется механизм, называемый динамическим SQL.

Суть этого механизма заключается в том, что запросы внутри блоков процедурного языка формируются в виде текстовых строк непосредственно во время работы программы, что позволяет получить более широкие возможности для работы с ними.

Процедурный язык Firebird предоставляет возможность выполнения динамических запросов в теле модуля с помощью оператора EXECUTE STATEMENT. SQL-запрос может быть:

- 1) селективным запросом, возвращающим набор данных;
- 2) DML-запросом, содержащим операторы добавления, изменения или удаления строк из таблиц;
- 3) вызовом ХП;
- 4) DDL-выражением, содержащим операторы создания, изменения или удаления объектов БД: доменов, таблиц, представлений, ХП. При этом нельзя выполнить оператор создания и удаления БД;
- 5) запросом, хранящим исходный код PSQL-блока.

В таком случае запрос записывается как <строка>, которая может быть сконструирована как локальная переменная в теле модуля или же передана внешним приложением или ХП в качестве входного аргумента для другой ХП. Оператор EXECUTE STATEMENT может применяться как в хранимых процедурах, так и в триггерах и выполнимых блоках на PSQL.

Синтаксис оператора EXECUTE STATEMENT:

```
[FOR] EXECUTE STATEMENT [(] <строка> [)] [(] [<входной_параметр1>
                                     [, <входной_параметр2> ...] [)]
[ON EXTERNAL [DATA SOURCE] <строка_соединения>]
[WITH { AUTONOMOUS | COMMON } TRANSACTION]
[AS USER user_name [PASSWORD user_password]]
[WITH CALLER PRIVILEGES]
[INTO: <имя_переменной1> [, : <имя_переменной2> ...]]
[DO
  <группа_операторов>]
```

где <строка> — строка, содержащая SQL-запрос, или переменная, которой присваивается значение <скалярный_подзапрос> (при этом <скалярный_подзапрос> может содержать параметры); <входной_параметр1>, <входной_параметр2> ... — присваивание значений параметрам в случае параметризованного SQL-запроса; <строка_соединения> — строка подключения к БД; AUTONOMOUS | COMMON — указывает, выполнять <группу_операторов> в ав-

тономной или текущей транзакции (если параметр не задан явно, выполнение <группы_операторов> происходит в текущей транзакции); user_name — имя пользователя, от имени которого будет выполнена <группа_операторов> (в качестве значения можно передавать контекстную переменную CURRENT_USER; если параметр не задан явно, то используется текущий пользователь); user_password — строка или переменная, содержащая пароль пользователя, от имени которого будет выполнена <группа_операторов>, для подключения к БД (если не указан параметр user_name или значение параметра user_name равно CURRENT_USER, то указание пароля можно пропустить); <имя_переменной1>, <имя_переменной2> ... — список переменных, куда необходимо вернуть значения в случае селективного запроса; <группа_операторов> — выполняется в случае циклического выполнения селективного запроса, может включать как один-единственный оператор, так и состоять из нескольких операторов, заключенных в операторные скобки BEGIN...END.

Порядок следования разделов оператора EXECUTE STATEMENT

```
[ON EXTERNAL [DATA SOURCE] <строка_соединения>]
[WITH { AUTONOMOUS | COMMON } TRANSACTION]
[AS USER user_name [PASSWORD user_password]]
[WITH CALLER PRIVILEGES]
```

не является жестким, т. е. разделы могут перечисляться в произвольном порядке. При этом повторное использование разделов в рамках одного оператора EXECUTE STATEMENT запрещено.

Можно выделить в синтаксисе Firebird три формы использования оператора EXECUTE STATEMENT.

Первая форма (простейшая) представлена синтаксисом

```
EXECUTE STATEMENT <строка>;
```

В простейшей форме оператор EXECUTE STATEMENT выполняет запрос SQL, записанный как <строка>, который производит какое-то действие, но не возвращает строк данных. Такой запрос может представлять собой следующее:

- 1) запрос DML (INSERT, DELETE, UPDATE);
- 2) EXECUTE PROCEDURE;
- 3) любой запрос DDL, за исключением CREATE DATABASE и DROP DATABASE.

Например, если создать процедуру в Firebird:

```
CREATE PROCEDURE One_Sample
  (Sql VARCHAR(60), Account VARCHAR(30))
AS
BEGIN
```

```
EXECUTE STATEMENT :Sql || :Account;
END;
```

то выполнить ее можно с помощью запроса

```
EXECUTE PROCEDURE One_Sample
('DELETE FROM REQUEST WHERE AccountCD =', '005488');
```

Вторая форма оператора EXECUTE STATEMENT имеет формат

```
EXECUTE STATEMENT <строка>
INTO :<имя_переменной1> [, :<имя_переменной2> ...]
```

В этом случае <строка> представляет собой запрос, возвращающий одиночную строку данных. Только <скалярный_подзапрос> может быть выполнен при использовании второй формы оператора EXECUTE STATEMENT (<подзапрос_столбца> и <табличный_подзапрос> не могут использоваться).

Пример создания выполняемой процедуры Firebird, использующей вторую форму синтаксиса оператора EXECUTE STATEMENT:

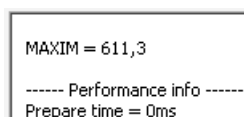
```
CREATE PROCEDURE Two_Sample
(ColName VARCHAR(50),
 TableName VARCHAR (50))
RETURNS (Maxim NUMERIC(15,2))
AS
BEGIN
EXECUTE STATEMENT 'SELECT MAX(' || :ColName || ') FROM '
|| :TableName INTO :MAXIM;
END;
```

Процедура Two_Sample определяет в указанной таблице БД максимальное значение в заданном столбце, имеющем числовой тип данных.

Процедура в Firebird может быть выполнена запросом

```
EXECUTE PROCEDURE Two_Sample ('Paysum', 'Paysumma');
```

Результат выполнения процедуры Two_Sample представлен на рис. 6.16.



```
MAXIM = 611,3
----- Performance info -----
Prepare time = 0ms
```

Рис. 6.16. Результат выполнения процедуры Two_Sample

Наконец, оператор EXECUTE STATEMENT поддерживает выполнение запроса SELECT внутри цикла FOR для возвращения в список переменных по одной строке за каждый проход цикла.

Третья форма представлена синтаксисом

```
FOR EXECUTE STATEMENT <строка>
INTO :<имя_переменной1> [, :<имя_переменной2> ...]
DO
    <группа_операторов>;
```

В данном случае <строка> может представлять собой запрос, возвращающий как одиночную строку данных, так и множество строк данных. Таким образом, любой запрос SELECT может быть использован в данной форме EXECUTE STATEMENT.

Рассмотрим процедуру, в которой применяется оператор EXECUTE STATEMENT в третьей форме синтаксиса. Пример создания процедуры:

```
CREATE PROCEDURE Three_Sample (TextField VARCHAR(10),
                               TableName VARCHAR(10),
                               AscDesc VARCHAR(4))
RETURNS (Line VARCHAR(20))
AS
DECLARE VARIABLE OneLine VARCHAR(50);
BEGIN
OneLine = 'SELECT ' || :TextField || ' FROM '
          || :TableName || ' ORDER BY 1 ';
IF (AscDesc = 'DESC') THEN OneLine = OneLine || 'DESC';
FOR
    EXECUTE STATEMENT :OneLine INTO :Line
DO
    SUSPEND;
END;
```

Данная процедура производит вывод значений столбца таблицы, имена которых задаются соответственно значениями 1-го и 2-го параметров, в порядке, определяемом значением 3-го параметра.

Процедура Three_Sample может быть выполнена с помощью запроса

```
SELECT * FROM Three_Sample ('Fio', 'Abonent', 'DESC');
```

Результатом такого выполнения является вывод значений столбца Fio таблицы Abonent, упорядоченных в обратном алфавитном порядке.

В Firebird 3.0 оператор EXECUTE STATEMENT поддерживает применение параметризованных запросов.

При использовании параметризованного запроса <строка>, содержащая текст этого запроса, должна быть заключена в круглые скобки. Например, в случае применения параметризованного запроса процедура One_Sample из приведенного выше примера может быть представлена таким образом:

```
CREATE PROCEDURE One_Sample
  (Sql VARCHAR(255), Account VARCHAR(30))
AS
BEGIN
  EXECUTE STATEMENT (:Sql)(Acc := :Account);
END;
```

а выполнить ее можно так:

```
EXECUTE PROCEDURE One_Sample
  ('DELETE FROM REQUEST WHERE AccountCD = :Acc', '005488');
```

В параметризованных запросах поддерживаются оба типа параметров: именованные и безымянные. Предыдущий пример демонстрировал использование запроса с именованными параметрами. В случае использования безымянных параметров предыдущая процедура может быть создана так:

```
CREATE PROCEDURE One_Sample
  (Sql VARCHAR(255), Account VARCHAR(30))
AS
BEGIN
  EXECUTE STATEMENT (:Sql)(:Account);
END;
```

а выполнить ее можно так:

```
EXECUTE PROCEDURE One_Sample
  ('DELETE FROM REQUEST WHERE AccountCD = ?', '005488');
```

При использовании параметризованных запросов необходимо учитывать ряд особенностей:

- одновременное использование именованных и безымянных параметров в одном и том же запросе запрещено;
- передача значений безымянным параметрам должна происходить в том же порядке, в каком они встречаются в тексте запроса;
- присваивание значений параметров должно осуществляться с помощью специального оператора «:=», аналогичного оператору присваивания в языке Pascal;
- если при присваивании значения параметру передается выражение (например, $P1 := :I + 1$), то значение этого выражения вычисляется только один раз.

Раздел оператора ON EXTERNAL DATA SOURCE позволяет выполнять cross-database запросы, т. е. запросы к другим БД из текущего подключения при соблюдении следующих условий:

- если при вызове оператора EXECUTE STATEMENT пропущен раздел ON EXTERNAL DATA SOURCE и пропущен раздел AS USER (либо присутствует раздел AS USER и параметр user_name равен

имени текущего пользователя CURRENT_USER), то вызываемый SQL-оператор выполняется в контексте текущего подключения и от имени текущего пользователя;

- если при вызове оператора EXECUTE STATEMENT пропущен раздел ON EXTERNAL DATA SOURCE, но при этом присутствует раздел AS USER и параметр user_name не равен имени текущего пользователя CURRENT_USER, то вызываемый SQL-оператор выполняется в контексте отдельного подключения к текущей БД. Иными словами, для выполнения SQL-оператора сервер создаст отдельное подключение к текущей БД. Данная особенность может быть полезна при необходимости выполнения SQL-операторов от имени другого пользователя;
- параметр <строка_соединения> раздела ON EXTERNAL DATA SOURCE имеет формат

```
<строка_соединения> : =  
[ {<имя_хоста> | <IP-адрес> } [ / <порт подключения> ] : <протокол> ]  
{ <имя_файла_БД> | <псевдоним_БД> }
```

Например, «localhost:/SQL/bases/abonents.fdb», «192.168.1.12/3052:ABONENTS», «D:\SQL/bases/abonents.fdb»;

- в случае подключения к внешним БД новое подключение использует ту же самую кодировку символов, что и у текущего подключения к БД. В некоторых случаях данное поведение сервера может служить источником ошибок, если кодировки символов, указанные при создании БД, отличаются.

Раздел WITH {AUTONOMOUS | COMMON} TRANSACTION позволяет выполнять SQL-оператор соответственно в рамках автономной или текущей транзакции при соблюдении нескольких условий:

- если оператор EXECUTE STATEMENT выполняется в контексте автономной транзакции, то автономная транзакция будет принудительно завершена при завершении текущей транзакции;
- если оператор EXECUTE STATEMENT выполняется в контексте автономной транзакции, то автономная транзакция будет подтверждена (commit) в случае успешного выполнения SQL-запроса или будет отменена (rollback) в случае неуспешного выполнения SQL-запроса;
- если оператор EXECUTE STATEMENT выполняется в контексте автономной транзакции, то автономная транзакция стартует с тем же уровнем изоляции данных, что и текущая, в контексте которой вызывается оператор EXECUTE STATEMENT;
- если раздел при вызове оператора EXECUTE STATEMENT пропущен, то оператор выполняется в рамках текущей транзакции.

Раздел AS USER позволяет задавать user_name, от имени которого будет выполнен SQL-запрос. Если раздел при вызове оператора

EXECUTE STATEMENT пропущен, то будет использоваться текущий пользователь CURRENT_USER.

Раздел WITH CALLER PRIVILEGES можно использовать в том случае, если оператор EXECUTE STATEMENT выполняется для текущего подключения к БД (раздел оператора ON EXTERNAL DATA SOURCE пропущен). Он позволяет выполнить SQL-запрос с привилегиями доступа к данным вызывающего PSQL-блока, ХП или триггера. Если раздел отсутствует, то EXECUTE STATEMENT выполняется с привилегиями пользователя, от имени которого был запущен PSQL-блок, но при этом не учитываются привилегии самого PSQL-блока, ХП или триггера. В некоторых случаях это требует выдачи дополнительных прав пользователям на объект БД, что может ставить под угрозу безопасность.

При использовании оператора EXECUTE STATEMENT в Firebird следует учесть, что невозможно подсчитать количество строк, возвращаемых выполненным в операторе запросом с помощью переменной ROW_COUNT. Контекстная переменная ROW_COUNT, использованная после оператора EXECUTE STATEMENT, всегда возвращает ноль.

Microsoft SQL Server поддерживает выполнение запросов, заданных строкой символов, с помощью оператора EXECUTE в формате [7]

```
{ EXEC | EXECUTE }
  ( { @строковая_переменная | [ N ] 'строка [ ? ]' } [ + ... ]
    [ , { значение | @переменная [ OUTPUT ] } [ ... ] ] )
[ AS { LOGIN | USER } = 'имя' ]
[ AT имя_связанного_сервера ];
```

Оператор выполняет командную строку символов, может также использоваться для отправки транзитных команд на связанные серверы и явно указывать контекст, в котором выполняется команда.

Строка может иметь тип NVARCHAR или VARCHAR. Если указано N, строка интерпретируется как NVARCHAR.

Секция AS применяется для указания контекста, в котором выполняется SQL-запрос. Ключевое слово LOGIN определяет выполнение от заданного имени входа, а USER — от заданного имени пользователя.

Секция AT используется при передаче транзитных команд связанным серверам, в этом же случае указывается передаваемое значение или переменная для сохранения аргумента с возможным признаком OUTPUT.

В простейшей форме оператор EXECUTE может быть реализован, как в ХП One_Sample:

```
CREATE PROCEDURE One_Sample
  (@Sql VARCHAR(60), @Account VARCHAR(30))
```

```
AS
BEGIN
    EXECUTE (@Sql + @Account);
END;
```

Выполнить созданную процедуру можно так:

```
EXECUTE One_Sample 'DELETE FROM REQUEST WHERE AccountCD = ', '005488';
```

Вторая форма оператора EXECUTE может быть реализована в MS SQL Server, как в ХП Two_Sample:

```
CREATE PROCEDURE Two_Sample
(@ColName VARCHAR(50),
 @TableName VARCHAR (50))
AS
BEGIN
    DECLARE @Maxim NUMERIC(15,2);
    EXECUTE ('SELECT MAX(' + @ColName + ') FROM ' + @TableName)
    RETURN (@Maxim);
END;
```

Выполнить процедуру можно, например, запросом

```
EXECUTE Two_Sample 'Paysum', 'Paysumma';
```

Третья форма оператора (для выполнения многострочного запроса) и параметризованные динамические запросы в MS SQL Server не поддерживаются.

Система управления БД Oracle поддерживает оператор EXECUTE IMMEDIATE для выполнения динамически формируемых запросов: SQL-запросов, которые возвращают одну строку, запросов обновления и изменения данных, DDL-запросов. Оператор имеет синтаксис [18]

```
EXECUTE IMMEDIATE <строка>
[INTO { <имя_переменной1> [, <имя_переменной2>]... | <запись>}]
[USING [IN | OUT | IN OUT] <параметр1> [, [IN | OUT | IN OUT]
<параметр2>]...]
[{RETURNING | RETURN} INTO <параметр1> [, <параметр2> ]...];
```

Секция INTO определяет переменные, в которые сохраняются выбранные значения столбцов, также можно использовать вместо переменных RECORD-тип, определенный пользователем, или %ROWTYPE запись, которая совпадает по формату с выбранной строкой.

Секция USING определяет параметры, которые передаются в динамический SQL-запрос. Параметры могут быть заданы выражениями. При выполнении запросов DML в секции USING задаются связывающие аргументы.

Секция RETURNING определяет параметры, в которые будут сохраняться значения, возвращаемые динамическим DML-запросом.

Рассмотрим примеры динамического SQL с оператором EXECUTE IMMEDIATE. Например, реализуем ранее описанную простейшую ХП One_Sample в Oracle:

```
CREATE PROCEDURE One_Sample
  (inSql IN VARCHAR2, Account IN VARCHAR2)
AS
BEGIN
  EXECUTE IMMEDIATE inSql || Account;
END;
```

Выполнить созданную процедуру можно путем вызова

```
EXEC One_Sample ('DELETE FROM REQUEST WHERE AccountCD = ',
  '005488');
```

Хранимая процедура Two_Sample, использующая оператор EXECUTE IMMEDIATE с возвратом параметра, в Oracle может быть реализована так:

```
CREATE PROCEDURE Two_Sample
  (ColName IN VARCHAR2, TableName IN VARCHAR2, Maxim OUT REAL)
AS
BEGIN
  EXECUTE IMMEDIATE 'SELECT MAX(' || ColName || ') FROM ' || TableName
  INTO Maxim;
END;
```

Выполнить процедуру можно с помощью анонимного PL/SQL блока (см. параграф 6.5):

```
DECLARE vMax NUMERIC(15,2);
BEGIN
  Two_Sample ('Paysum', 'Paysumma', vMax);
END;
```

В Oracle, в отличие от Firebird, после выполнения запросов оператором EXECUTE IMMEDIATE можно получить количество строк с помощью переменной SQL %ROWCOUNT.

В заключение следует отметить, что операторы выполнения динамических запросов добавляют гибкости модулям на процедурных языках, но с риском ошибок. Такие операторы следует применять только в случае невозможности получить нужные результаты другими средствами или когда это действительно улучшает выполнение запроса (что маловероятно).

При использовании оператора динамического выполнения нужно помнить о его особенностях:

- во время компиляции синтаксический анализатор не может проверить синтаксис запроса в строке аргумента;
- во время компиляции возвращаемые значения не проверяются на соответствие типов данных (например, строка '1234' может быть преобразована в целое число, а строка 'абв' вызовет ошибку преобразования), поэтому они должны быть внимательно проверены разработчиком, чтобы избежать непредсказуемых исключений преобразования данных при последующем выполнении процедуры;
- не проверяются зависимости или существование защиты для предотвращения удаления или изменения таблиц и столбцов;
- операции динамического выполнения работают медленно, потому что встроенный оператор должен подготавливаться на сервере каждый раз перед выполнением.

6.3. Хранимые функции

В различных СУБД, поддерживающих процедурные языки, существует возможность создавать с их использованием не только ХП, но также и функции. Процедурная (хранимая) функция представляет собой подпрограмму, которая принимает входные параметры, выполняет числовые вычисления, результат которых возвращается в виде значения функции [6, 15, 16]. Важно отметить, что, в отличие от ХП, скалярная функция может быть указана в любом месте выражения, в том числе в вычисляемых столбцах и определениях ограничений CHECK.

В Firebird хранимые PSQL-функции, в отличие от селективных ХП, не могут возвращать табличное значение. Синтаксис определения функции следующий:

```
{ CREATE [OR ALTER] | ALTER | RECREATE } FUNCTION имя_функции
  [(<список_входных_параметров>)]
  [RETURNS (<тип данных>)] [DETERMINISTIC]
  { EXTERNAL NAME '<внешнее_имя>' ENGINE <имя_движка> |
    AS <тело_функции>}
```

где <список_входных_параметров> аналогичен списку для ХП, а <тело_функции> аналогично <тело_процедуры>, но дополнительно использует оператор RETURN для возврата значения функции. Возвращаемое значение представляет собой скалярное значение указанного типа данных.

Опция DETERMINISTIC указывает, что функция детерминированная [15]. Детерминированные функции каждый раз возвращают один и тот же результат при одинаковом наборе входных значений, используя значения из кэша метаданных (если они там есть). Недетерминированные функции могут возвращать каждый раз разные результаты при одном и том же наборе входных значений.

Хранимая функция может быть расположена во внешнем модуле. В этом случае вместо тела функции указывается место расположения функции во внешнем модуле с помощью секции EXTERNAL NAME. В качестве внешнего имени через разделитель указываются имя внешнего модуля, имя функции внутри модуля и определенная пользователем информация. После ключевого слова ENGINE указывается имя движка для обработки подключения внешних модулей (в Firebird — движок UDR).

Примечание. Не следует путать внешние функции, известные как UDF и объявленные с помощью DECLARE EXTERNAL FUNCTION, с функциями, расположенными во внешних модулях, называемыми UDR (User Defined Routine) и объявленными с помощью CREATE FUNCTION ... EXTERNAL NAME. Первые являются унаследованными из предыдущих версий Firebird, и их возможности существенно уступают возможностям нового типа внешних функций.

Для примера преобразуем ХП Factorial, вычисляющую факториал числа (см. подпараграф 6.2.4), в функцию fFactorial. Запрос на определение такой функции более компактный, чем определение аналогичной ХП, и будет выглядеть таким образом:

```
CREATE FUNCTION fFactorial (Num INTEGER)
RETURNS DOUBLE PRECISION
AS
BEGIN
    IF (Num < 0) THEN RETURN NULL;
    IF (Num = 1 OR Num = 0) THEN
        RETURN 1;
    ELSE
        RETURN (fFactorial (Num - 1) * Num);
    WHEN ANY DO RETURN NULL;
END;
```

Хранимые функции в Firebird вызываются так же, как и обычные внутренние функции. Например, созданную функцию fFactorial можно вызвать так:

```
SELECT fFactorial (170) FROM RDB$DATABASE;
```

Результат выполнения запроса будет аналогичен результату, представленному на рис. 6.10, б.

Microsoft SQL Server тоже предоставляет возможность создания определяемых пользователем функций. В упрощенном виде формат определения функций такой [7]:

```
CREATE FUNCTION имя
( [ @параметр [AS] тип_данных [ = значение_по_умолчанию ]
```



```

    [ READONLY ] [ ,... ] ] )
RETURNS тип_данных
[ WITH <опция> [ ,... ] ]
[AS]
BEGIN
    <тело_функции>
    RETURN <скалярное_выражение>
END [;]

```

Опция READONLY указывает, что параметр не может быть обновлен или изменен при определении функции. Если тип параметра является определяемым пользователем табличным типом, то опция READONLY должна быть задана.

Секция RETURNS задает тип данных возвращаемого функцией значения, которое может быть скалярным значением или таблицей.

Рассмотрим в качестве примера, как ХП вычисления факториала может быть преобразована в MS SQL Server в функцию. Скрипт создания функции:

```

CREATE FUNCTION dbo.fFactorial(@Num INTEGER)
RETURNS INTEGER
AS
BEGIN
    DECLARE @N_Factorial INTEGER;
    IF (@Num < 0 OR @Num > 170) RETURN NULL;
    IF (@Num = 1 OR @Num = 0)
        SET @N_Factorial = 1;
    ELSE
        SET @N_Factorial = dbo.fFactorial(@Num - 1) * @Num;
    RETURN (@N_Factorial );
END

```

Следует обратить внимание, что функция создается и вызывается с префиксом dbo перед именем. Префикс задает имя владельца функции — Database Owner (dbo). По правилам MS SQL Server скалярные функции должны вызываться с помощью как минимум двухкомпонентного имени.

Последним оператором, входящим в функцию в MS SQL Server, должен быть оператор RETURN.

Функция может быть не только указана в любом месте SQL-выражения, но также выполнена оператором EXECUTE. Функция, возвращающая табличное значение, может быть вызвана в любом месте, где допускаются табличные выражения: в секции FROM запросов SELECT, INSERT, UPDATE и DELETE.

Созданная функция dbo.fFactorial может быть вызвана, как и обычная встроенная функция, в запросе

```

SELECT dbo.fFactorial(5)

```

Рекурсивный вызов функцией самой себя выполняется путем присваивания возвращаемого ею значения переменной TSQL. Возврат значения осуществляется с помощью оператора RETURN.

Система управления БД Oracle поддерживает создание PL/SQL-функций в формате

```
CREATE [ OR REPLACE ] FUNCTION имя_функции
  [ (<параметр> [ IN | OUT | IN OUT] <тип_данных> [,...]) ]
RETURN <тип_данных>
{ IS | AS }
BEGIN
  <тело_функции>
[EXCEPTION <блок_исключений>]
END [ имя_функции ];
```

Например, функция вычисления факториала в Oracle может быть реализована так:

```
CREATE FUNCTION fFactorial (Num IN INTEGER)
RETURN DOUBLE PRECISION
IS
BEGIN
  IF (Num < 0 OR Num > 170) THEN
    RETURN NULL;
  END IF;
  IF (Num = 1 OR Num = 0) THEN
    RETURN 1;
  ELSE
    RETURN (fFactorial (Num - 1) * Num);
  END IF;
END fFactorial;
```

В подпараграфе 6.2.4 приведен пример получения многострочного результата на выходе ХП ListAbonent в Oracle и упоминалось, что аналогичного результата можно добиться с использованием *pipelined*-функции.

Конвейерные (*pipelined*) функции на выходе выдают набор строк, при этом их принципиальное отличие от других функций в том, что это множество строк они выдают не сразу, а по одной [18, 22]. Внутри такой функции используется оператор PIPE ROW, который выдает одну строку результата и приостанавливает выполнение функции до тех пор, пока среда, вызвавшая эту функцию (это может быть, например, тоже конвейерная функция), не потребует следующую строку. Вызов оператора RETURN в таких функциях «пустой», поскольку результат возвращается клиенту немедленно при вызове оператора PIPE ROW и нет никакой коллекции, возвращаемой из функции.

Конвейерная обработка отменяет надобность в создании огромных наборов, передавая строки из функции по мере их создания,

сохраняя память и позволяя запустить последующую обработку еще до окончания генерации всех строк.

Рассмотрим пример конвейерной функции, которая возвращает такую же информацию, как приведенная в подпараграфе 6.2.4 ХП ListAbonent, только без записи в таблицу.

В первую очередь, необходимо объявить типы для приемника данных. Тип для отдельной записи:

```
CREATE OR REPLACE TYPE PayInfo_Type AS OBJECT
(AccountCD VARCHAR2(6), Fio VARCHAR2(30),
 PayDate DATE, PaySum NUMERIC(15,2),
 PayMonth SMALLINT, PayYear SMALLINT);
```

и табличный тип:

```
CREATE OR REPLACE TYPE PayInfo_Table IS TABLE OF PayInfo_Type;
```

Саму конвейерную функцию создадим с помощью следующего скрипта:

```
CREATE FUNCTION fListAbonent (Mes IN INTEGER, God IN INTEGER)
RETURN PayInfo_Table PIPELINED
AS
  CURSOR Pay_Cur IS
    SELECT A. AccountCD, A.Fio, P. PayDate,
           P.PaySum, P. PayMonth, P. PayYear
    FROM Abonent A, PaySumma P
    WHERE A. AccountCD = P. AccountCD
          AND P.PaySum > 500
          AND P.PayMonth = Mes
          AND P.PayYear = God;
BEGIN
  FOR I IN Pay_Cur LOOP
    PIPE ROW (PayInfo_Type(i. AccountCD, i.Fio, i.PayDate,
                           i.PaySum, i.PayMonth, i.PayYear));
  END LOOP;
  RETURN;
END fListAbonent;
```

Функция создается с признаком PIPELINED и указывается, что она возвращает результат предварительно созданного табличного типа PayInfo_Table. Объявляется курсор Pay_Cur и затем обрабатывается в цикле, при этом текущая строка возвращается оператором PIPE ROW в вызывающую среду. Оператор RETURN вызывается без аргументов, так как все результаты функция уже вернула через PIPE ROW.

Использовать созданную функцию для получения результатов можно в секции FROM запроса SELECT так:

```
SELECT * FROM Table(fListAbonent (10,2018));
```

В Firebird и Oracle удалить созданную и неиспользуемую функцию можно запросом

```
DROP FUNCTION имя_функции;
```

В MS SQL Server можно одним запросом удалить сразу несколько функций:

```
DROP FUNCTION имя_функции1, [имя_функции2 ...];
```

6.4. Процедурные пакеты

Во многих современных СУБД существует возможность создания пакетов на процедурном языке. В некоторых СУБД пакеты создаются явно и представляют собой отдельный объект БД, в других — нет.

Пакеты в Firebird представляют собой объект БД в виде набора функций и процедур, управляемых как единое целое и предназначенных для решения задач в рамках одного модуля (подсистемы) информационной системы [6, 15]. PSQL-пакеты состоят из двух частей: заголовка (ключевое слово PACKAGE) и тела (ключевое слово PACKAGE BODY). Такое разделение очень похоже на устройство модулей Delphi, где заголовок соответствует интерфейсной части (Interface), а тело — исполняемой части (Implementation). Сначала создается заголовок (CREATE PACKAGE), а затем тело (CREATE PACKAGE BODY).

Заголовки пакетов создаются с использованием синтаксиса

```
{CREATE [OR ALTER] | ALTER | RECREATE} PACKAGE <имя_пакета>
AS
BEGIN
    [<объявление_функции>...]
    [<объявление_процедуры>...]
END;
```

где

```
<объявление_функции> ::=
    FUNCTION <имя_функции> [( <список_входных_параметров> )]
    RETURNS <тип_данных>;
<объявление_процедуры> ::=
    PROCEDURE <имя_процедуры> [( < список_входных_параметров > )]
    [RETURNS ( < список_выходных_параметров > )];
```

Тело пакета создается с использованием синтаксиса

```
{CREATE | RECREATE} PACKAGE BODY <имя_пакета>
AS
BEGIN
```

```

[<объявление_функции>...]
[<объявление_процедуры>...]
[ <функция> ... ]
[ <процедура> ... ]
END;

```

где

```

<функция> ::=
{FUNCTION <имя_функции> [( <список_входных_параметров > )]
  RETURNS <тип_данных>
  AS
  BEGIN
  ...
  END; |
  FUNCTION <имя_функции> [( <список_входных_параметров > )]
  RETURNS <тип_данных>
  EXTERNAL NAME '<имя_внешней_функции>'
  ENGINE <внешний_модуль>;
}
<процедура> ::=
{ PROCEDURE <имя_процедуры> [( < список_входных_параметров > )]
  [RETURNS ( < список_выходных_параметров > )]]
  AS
  BEGIN
  ...
  END; |
  PROCEDURE <имя_процедуры> [( < список_входных_параметров > )]
  [RETURNS ( < список_выходных_параметров > )]]
  EXTERNAL NAME '<имя_внешней_функции>'
  ENGINE <внешний_модуль>;
}

```

Тело пакета должно содержать реализацию всех процедур, объявленных в заголовке пакета. Значения по умолчанию для параметров процедур указываются при объявлении процедуры в заголовке пакета и не могут быть переопределены в теле. Исключение составляют приватные процедуры, которые определяются в теле пакета и доступны для использования только внутри данного пакета.

Запуск (выполнение) процедур/функций из пакета выполняется в виде

```
EXECUTE PROCEDURE <имя_пакета>.{<функция>|<процедура>;}
```

В СУБД Oracle **пакет** — это объект БД, который группирует процедуры, функции и другие программные объекты PL/SQL [2, 18]. Пакеты в Oracle, как и в Firebird, обычно состоят из двух частей, заголовка и тела, хотя иногда в теле нет необходимости. Заголовок пакета объявляет типы, переменные, константы, исключения, курсоры и подпрограммы, доступные для использования в пакете.

Тело пакета полностью определяет (реализует) курсоры и подпрограммы. В отличие от ХП и функций, пакеты нельзя вызывать, передавать им параметры или вкладывать их друг в друга. В остальном формат создания пакета похож на формат определения других процедурных модулей:

```
CREATE [OR REPLACE] PACKAGE <имя_пакета>
[AUTHID { CURRENT_USER | DEFINER }]
{IS | AS}
    <список_объявления>
END [<имя_пакета>];
```

где

```
<список_объявления> ::=
    [<объявление_типа>...]
    [<объявление_переменной>...]
    [<объявление_константы>...]
    [<объявление_исключения>...]
    [<объявление_курсора>...]
    [<объявление_функции>...]
    [<объявление_процедуры>...];
CREATE [OR REPLACE] PACKAGE BODY <имя_пакета>
{IS | AS}
    [<список_объявления>]
    [ <курсор> ... ]
    [ <функция> ... ]
    [ <процедура> ... ]
END [<имя_пакета>];
```

Запрос CREATE OR REPLACE позволяет создать пакет при его отсутствии либо полностью пересоздать пакет.

Опция AUTHID при создании заголовка пакета позволяет указать привилегии, с которыми будет выполняться пакет: либо с привилегиями текущего пользователя (CURRENT_USER), либо с привилегиями владельца текущей схемы БД (DEFINER).

Перекомпилировать пакет в Oracle можно запросом ALTER PACKAGE формата

```
ALTER PACKAGE <имя_пакета>
COMPILE
{PACKAGE | SPECIFICATION | BODY}
[<параметр_компиляции1> = <значение>
    [, <параметр_компиляции2> = <значение>] ...]
[REUSE SETTINGS];
```

Опция COMPILE указывает, что необходимо перекомпилировать пакет (PACKAGE), заголовок (SPECIFICATION) или тело (BODY).

Опция REUSE SETTINGS позволяет сохранять начальные настройки и использовать их при перекомпиляции.

Запуск процедур/функций из пакета выполняется в виде

```
EXECUTE PROCEDURE <имя_пакета>.{<функция>|<процедура>};
```

В MS SQL Server отсутствуют возможности явного создания и использования процедурных пакетов. **Пакет** в MS SQL Server — это группа из одной или нескольких инструкций языка T-SQL, отправляемых одновременно на SQL Server для выполнения [7]. Пакеты разделяются ключевым словом GO. При использовании пакетов применяются следующие правила:

- запросы CREATE DEFAULT, CREATE FUNCTION, CREATE PROCEDURE, CREATE RULE, CREATE SCHEMA, CREATE TRIGGER и CREATE VIEW не могут быть объединены с другими запросами в пакете. Запрос CREATE должен быть первым запросом в пакете. Все последующие запросы в этом пакете рассматриваются как часть определения первого запроса CREATE;
- в одном и том же пакете нельзя сначала изменить таблицу и затем обратиться к новым столбцам;
- если запрос EXECUTE — первый запрос в пакете, то ключевое слово EXECUTE не требуется.

Использование пакетов в процедурных языках предназначено для выполнения следующих задач.

1. *Модульность.* Идея состоит в том, чтобы разделить блоки взаимозависимого кода на логические модули, как это обычно делается в языках программирования. В обычных языках программирования группировка кода различными способами (пространство имен, модули или классы) позволяет упростить решение множества задач. Со стандартными ХП и функциями в БД добиться такого же результата довольно сложно. Стандартные ХП и функции могут быть сгруппированы в различные скрипты, однако скрипты не сохраняются в БД как часть метаданных, в отличие от процедурных пакетов.

2. *Упрощение отслеживания зависимостей.* Пакеты предоставляют механизм для облегчения отслеживания зависимостей между набором процедур, а также между самим пакетом и другими процедурами, как упакованными в пакеты, так и неупакованными.

Разделение пакета на заголовок и тело приводит к тому, что каждый раз, когда пакетная процедура использует определенный объект БД, зависимость от этого объекта регистрируется в системных таблицах СУБД. После этого для удаления или изменения данного объекта, в первую очередь, необходимо удалить то, что зависит от него. От объекта зависит тело пакета, и это тело может быть удалено, даже если другие объекты БД зависят от этого пакета. Это возможно благодаря тому, что удаляется только тело пакета, а заголовки остаются, позволяя воссоздать тело после того, как изменения, потребовавшие удаления, выполнены.

3. *Упрощение управления полномочиями.* В соответствии с политикой безопасности создание и выполнение любых процедур должно выполняться пользователем, имеющим соответствующие привилегии. При объединении процедур в пакеты нет необходимости раздавать права на отдельные процедуры: полномочия раздаются на пакет в целом и действуют на все процедуры тела пакета. При этом привилегии на все используемые пакетом объекты предоставляются процедурному пакету, а не самому пользователю (роли). Пользователю (роли) предоставляются полномочия на пакет. Например, чтобы предоставить в Firebird роли Role_Ab привилегии на пакет Paket_Ab, который использует таблицу Abonent, необходимо выполнить запросы:

```
GRANT SELECT ON TABLE Abonent TO PACKAGE Paket_Ab;  
GRANT EXECUTE ON PACKAGE Paket_Ab TO ROLE Role_Ab;
```

4. *Создание «приватной области».* Все языки программирования имеют понятие «область действия процедуры». Это означает, что процедуры могут использоваться только в определенном месте программного кода. Процедурные пакеты работают так же, как модули Delphi в этом отношении. Если процедура не объявлена в заголовке пакета (интерфейсная часть) и реализуется в теле, то она становится приватной. Приватную процедуру можно вызвать только внутри своего пакета.

Рассмотрим пример создания процедурного пакета Paket_Factorial, который позволяет вычислять факториал для заданного диапазона чисел. Пакет Paket_Factorial объединит процедуру FactorialSet (см. подпараграф 6.2.4) и функцию fFactorial (см. подпараграф 6.3). Скрипт создания пакета в Firebird:

```
SET TERM !! ;  
CREATE OR ALTER PACKAGE Paket_Factorial  
AS  
BEGIN  
    -- общедоступная процедура  
    PROCEDURE FactorialSet (Minf INTEGER, Maxf INTEGER);  
END !!  
CREATE PACKAGE BODY Paket_Factorial  
AS  
BEGIN  
    -- приватная функция  
    FUNCTION fFactorial (Num INTEGER) RETURNS DOUBLE PRECISION;  
    PROCEDURE FactorialSet (Minf INTEGER, Maxf INTEGER)  
    AS  
        DECLARE VARIABLE i INTEGER;  
        DECLARE VARIABLE f DOUBLE PRECISION;  
    BEGIN  
        DELETE FROM Ftable;
```



```

        IF (Maxf < Minf) THEN EXIT;
        i = Minf;
        WHILE ( i <= Maxf) DO
        BEGIN
            f = fFactorial (:i);
            INSERT INTO Ftable (Fnum, Fvalue) VALUES (:i, :f);
            i = i + 1;
        END
    END
    FUNCTION fFactorial (Num INTEGER) RETURNS DOUBLE PRECISION
    AS
    BEGIN
        IF (Num < 0 OR Num > 170) THEN RETURN NULL;
        IF (Num = 1 OR Num = 0) THEN
            RETURN 1;
        ELSE
            RETURN (fFactorial (Num -1) * Num);
        END
    END !!
    SET TERM ; !!

```

В данном скрипте сначала создается заголовок пакета Paket_Factorial, который содержит объявление общедоступной (public) процедуры FactorialSet. Затем создается тело пакета, которое содержит приватную функцию fFactorial, доступную только внутри данного пакета, и тело процедуры FactorialSet. Тело пакета содержит весь программный код по вычислению значений факториала для заданного диапазона чисел. Результат записывается в таблицу Ftable (предполагается, что она предварительно создана, см. подпараграф 6.2.4).

Пример вызова пакета может быть таким

```
EXECUTE PROCEDURE Paket_Factorial.FactorialSet (1,5);
```

Результат выполнения пакета совпадает с результатом, представленным на рис. 6.11.

В Oracle аналогичный скрипт создания пакета будет выглядеть так:

```

CREATE OR REPLACE PACKAGE Paket_Factorial
IS
    PROCEDURE FactorialSet (Minf INTEGER, Maxf INTEGER);
END Paket_Factorial;
CREATE OR REPLACE PACKAGE BODY Paket_Factorial
IS
    FUNCTION fFactorial (Num INTEGER) RETURN DOUBLE PRECISION;
    PROCEDURE FactorialSet (Minf INTEGER, Maxf INTEGER)
    IS
        i INTEGER;
        f DOUBLE PRECISION;

```

```

BEGIN
  DELETE FROM Ftable;
  IF (Maxf < Minf) THEN
    RETURN;
  END IF;
  i := Minf;
  WHILE ( i <= Maxf) LOOP
    Factorial (i, f);
    INSERT INTO Ftable (Fnum, Fvalue) VALUES (i, f);
    i := i + 1;
  END LOOP;
END FactorialSet;
FUNCTION fFactorial (Num IN INTEGER)
RETURN DOUBLE PRECISION
IS
BEGIN
  IF (Num < 0 OR Num > 170) THEN
    RETURN NULL;
  END IF;
  IF (Num = 1 OR Num = 0) THEN
    RETURN 1;
  ELSE
    RETURN (fFactorial (Num - 1) * Num);
  END IF;
END fFactorial;
END Paket_Factorial;
/

```

Удаление заголовка пакета как в Firebird, так и в Oracle, выполняется с использованием синтаксиса

```
DROP PACKAGE <имя_пакета>;
```

Удаление тела пакета:

```
DROP PACKAGE BODY <имя_пакета>;
```

6.5. Выполнимые блоки

При работе с некоторыми СУБД существует возможность выполнения кода на процедурном языке без оформления именованных ХП или функций. В Firebird такую возможность предоставляет выполнимый неименованный (анонимный) блок (PSQL-блок), который имеет синтаксис

```

EXECUTE BLOCK [(входной_параметр1 <тип_данных> = : входной_параметр1
                                                         [, входной_параметр2 <тип_данных> = :
входной_параметр2])]
[RETURNS (выходной_параметр1 <тип_данных>
                                                         [, выходной_параметр2 <тип_данных> ...])]
AS <тело_блока>

```

Такой фрагмент кода на PSQL подобен ХП (может иметь входные и выходные параметры, результат работы идентичен результату ХП). PSQL-блок может использоваться в клиентских приложениях для реализации требуемой логики работы системы, когда в используемой БД нет необходимой именованной ХП или функции. Входные параметры в таком случае передаются в PSQL-блок из приложения, а выходные возвращаются клиентскому приложению. В IBExpert PSQL-блок можно выполнить в SQL-редакторе как обычный запрос.

Выполнимый блок <тело_блока> может включать в себя все те же конструкции (операторы, запросы), что и тело ХП или функции.

При изучении хранимых процедур была рассмотрена процедура Three_Sample, выполняющая вывод значений столбца таблицы, имена которых задаются соответственно значениями 1-го и 2-го параметров, упорядоченными в порядке, определяемом значением 3-го параметра. Реализовать ту же самую операцию без создания именованной ХП в Firebird можно следующим выполнимым блоком:

```
EXECUTE BLOCK (TextField VARCHAR(10) = :TextField,
TableName VARCHAR(10) = :TableName, AscDesc VARCHAR(4) = :AscDesc)
RETURNS (Line VARCHAR(20))
AS
    DECLARE VARIABLE OneLine VARCHAR(50);
    BEGIN
    Oneline = 'SELECT ' || :TextField || ' FROM ' || :TableName || ' ORDER
BY 1 ';
    if (AscDesc='DESC') then OneLine=Oneline || 'DESC';
    FOR
        EXECUTE STATEMENT :OneLine INTO :Line
    DO
        SUSPEND;
    END;
```

Результат выполнения блока для столбца Fio таблицы Abonent совпадает с результатом вызова ХП Three_Sample.

В Oracle код на PL/SQL может быть выполнен с помощью анонимного блока. Задача создания неименованных блоков в Oracle фактически сводится к написанию SQL-операторов, соблюдая при этом структуру разделов, общую для всех модулей на PL/SQL:

```
[<список_объявления>]
BEGIN
    <блок_исполнения>
[EXCEPTION
    <блок_обработки_исключений>]
END;
/
```

Входных и выходных параметров неименованные блоки в Oracle не имеют. Косая черта после окончания блока указывает Oracle, что после загрузки блока необходимо скомпилировать код.

Примеры использования анонимных блоков в Oracle были приведены в подпараграфах 6.2.4 и 6.2.5 при вызове ХП с входными и выходными параметрами.

6.6. Триггеры

Триггер — это особый вид процедуры обработки события, вызываемый автоматически без непосредственного участия пользователя. В большинстве случаев реализуются триггеры для обработки событий, связанных с таблицей БД [8].

Триггеры — важнейший механизм для активных баз данных, которые являются не пассивными системами хранения, а активно реагируют на изменения в данных путем генерации различных событий и их обработки. Триггеры используются для решения следующих задач:

- 1) реализация на сервере бизнес-логики в рамках концепции активных баз данных;
- 2) реализация динамических ограничений целостности;
- 3) ведение журналов аудита действий с данными;
- 4) автоматизация администрирования баз данных.

Триггеры применяются, как правило, для автоматической поддержки сложных правил целостности и дополнительной защиты БД. Стандартные возможности защиты данных — роли и привилегии — задают, какие именно команды языка DML может выполнять конкретный пользователь над конкретной таблицей. В дополнение к этому триггер может ограничить операции DML, например, в определенные дни недели.

Кроме того, триггеры часто используются в следующих случаях:

- автоматическая генерация выводимых значений столбцов;
- защита от неправильных транзакций;
- проверка допустимости вводимых данных;
- обеспечение сложных проверок прав доступа;
- задание сложных правил целостности;
- фиксирование и обработка различных событий в прозрачном для пользователя режиме;
- обеспечение сложного протоколирования;
- обеспечение синхронной репликации таблиц;
- сбор статистики о доступе к таблицам.

Все современные СУБД поддерживают триггеры, реагирующие на события изменения данных в таблицах БД, но, кроме того, некоторые СУБД поддерживают особые виды триггеров.

В СУБД Firebird существует три вида триггеров: триггеры DML, триггеры DDL и триггеры БД.

Триггеры DML исполняются СУБД автоматически при проведении операций изменения данных. Триггер DML фактически является

обработчиком события на изменение данных в базе. При создании такой триггер связывается с одной таблицей или представлением БД. Триггеры не могут быть созданы для временных или системных таблиц, хотя они и могут обращаться к временным таблицам. Если в теле триггера необходимо обратиться к данным системных таблиц, то можно создать представление, которое будет отображать информацию из системных таблиц, и обращаться в триггере к этому представлению.

Триггеры DML предоставляют возможность производить некоторые определяемые пользователем действия при выполнении над данными заданной таблицы или представления трех операций: удаления, изменения и вставки. Один и тот же триггер может отслеживать разные виды операций (например, вставку и изменение), причем операцию можно отслеживать до и после ее выполнения.

Триггеры DDL активируются в ответ на различные события языка DDL. Если триггеры DML реагируют на изменение данных в базе, то триггеры DDL реагируют на изменение схемы БД. Они предоставляют возможность производить определяемые пользователем действия при выполнении операций создания, изменения и удаления объектов БД.

Триггеры БД не связываются с конкретной таблицей БД. Они исполняются при подключении к БД или отключении от нее, а также при запуске, успешном завершении или откате транзакции.

В MS SQL Server также существуют три вида триггеров, которые отличаются по функциям и по синтаксису создания и изменения [4, 23].

Триггеры DML вызываются при выполнении запросов INSERT, UPDATE или DELETE. Допускается создание триггера, реагирующего на два или на все три запроса. По времени вызова триггеры разделяются на AFTER и INSTEAD OF. Триггер AFTER выполняется после успешного завершения вызвавшего его события. Можно определить несколько AFTER-триггеров для каждой операции (INSERT, UPDATE, DELETE). По умолчанию в SQL Server все триггеры являются AFTER-триггерами. Триггер INSTEAD OF вызывается вместо выполнения команд и, в отличие от AFTER-триггера, может быть определен как для таблицы, так и для представления. Для каждой операции INSERT, UPDATE, DELETE можно определить только один INSTEAD OF-триггер.

Триггеры DDL реагируют на события изменения схемы БД: создание, изменение или удаление отдельных объектов БД (таблиц, индексов, пользовательских типов данных и т. д.).

Триггеры входа в систему запускаются при соединении пользователя с экземпляром сервера. Их можно применять для дополнительной проверки полномочий пользователей.

Система управления БД Oracle также поддерживает несколько видов триггеров [8].

Триггеры DML вызываются событиями изменения данных. При этом, в отличие от других СУБД, Oracle позволяет создавать триггеры событий на уровне строки и триггеры событий на уровне операции. Вариантов срабатывания триггера три:

- 1) перед выполнением события, вызывающего срабатывание (BEFORE);
- 2) после выполнения события, вызывающего срабатывание (AFTER);
- 3) вместо выполнения события, вызывающего срабатывание (INSTEAD OF).

Комбинирование первых двух вариантов с заданием срабатывания для каждой строки или всего запроса дает четыре возможных реализации триггера: перед запросом, перед строкой, после запроса и после строки.

В версии Oracle Database 11g появилось понятие *составного триггера* (COMPOUND), который позволяет определить в триггере разные участки для разных моментов срабатывания. Составные триггеры повышают производительность, поскольку загружать триггер приходится только один раз.

Среди *триггеров DML* отдельно можно выделить триггеры INSTEAD OF, позволяющие реализовать операции манипулирования данными для представлений, которые естественным образом их не допускают (например, для обновления представлений, ссылающихся на столбцы из нескольких базовых таблиц). Применять триггеры INSTEAD OF следует с осторожностью из-за многочисленных потенциальных проблем, связанных с модификацией данных в базовых таблицах представления.

Триггеры для системных событий (триггеры для событий уровня БД). Например, можно определить триггер для таких системных событий, как запуск и останов БД.

Триггеры для пользовательских событий (триггеры для событий уровня схемы). Например, можно определить триггер для таких пользовательских событий, как вход в систему и выход из нее.

Использование триггеров в любой СУБД имеет ряд преимуществ:

- возможность автоматической проверки триггерами DML корректности данных, определенные значения которых необходимо хранить в соответствующей таблице;
- создание более гибкой системы поддержки целостности БД, которую невозможно реализовать декларативно;
- сокращение объема поддержки приложений, поскольку изменения триггеров автоматически отражаются во всех приложениях, использующих связанные с триггерами таблицы (без необходимости их повторной сборки и компиляции);

- возможность контролировать информацию посредством создания системы автоматической фиксации изменений данных в таблицах БД, схемы БД и пользователей, производящих эти изменения (ведение журнала изменений);
- автоматически оповещать другие программы о том, что необходимо делать в случае изменения информации, содержащейся в таблице;
- публиковать информацию о различных событиях.

6.6.1. Триггеры DML

Триггеры DML применяются для обеспечения ссылочной целостности, целостности данных и целостности, определяемой пользователем, т. е. бизнес-правил. С точки зрения информационной системы: **«бизнес-правило** — это указание, определяющее или ограничивающее определенный аспект бизнеса». Его назначение — защитить структуру бизнеса, контролировать или влиять на его операции. Бизнес-правила представляют собой условия того, что данные соответствуют предметной области.

Бизнес-правила реализуются как специализированный вид логики, описывающей ограничения на образ действий, которые система или люди должны учитывать в своем поведении. Эти правила определяются целым рядом факторов, изменяются от страны к стране, от отрасли к отрасли, от бизнеса к бизнесу. Бизнес-правила существуют на разных уровнях, могут значительно различаться по размерам области влияния, но, несмотря на это, все бизнес-правила имеют одно общее свойство: они управляют некоторой составляющей бизнеса. Их назначение — защитить структуру бизнеса, контролировать или влиять на его операции.

Ограничения могут быть статическими или динамическими. *Статические ограничения* должны выполняться для каждого состояния БД. *Динамические ограничения* проявляются при переходе БД из одного состояния в другое. Например, исполнителю в день не может быть назначено более трех ремонтных заявок или при повышении заработной платы слесарю новое значение должно быть больше старого.

Не следует вставлять, редактировать или удалять строки в таблице, если еще не реализован комплекс бизнес-правил. Бизнес-правила определяют реакцию системы на добавление, изменение или удаление данных, обеспечивая непротиворечивость и ссылочную целостность БД. Бизнес-правила разрабатываются на основе тщательного изучения автоматизируемой области человеческой деятельности.

Таким образом, чаще всего триггеры DML используются:

- для обеспечения непротиворечивости данных, например для выполнения действий по отношению к другим таблицам, основан-

ных на изменениях, произведенных в какой-то таблице, например, для внесения записи в контрольный журнал с целью регистрации каждого случая обновления или удаления начислений или оплат;

- для выполнения дополнительной проверки и отмены ввода данных, например, чтобы удостовериться в том, что разрешенная для абонента сумма задолженности не превышена, в противном случае операция блокируется;

- для подсчета значений вычисляемых полей или обновления меток даты/времени, например, расчета значения задолженности абонента.

6.6.1.1. Определение триггера

С любым событием, вызывающим изменение содержимого таблицы (представления), можно связать определенные действия, которые СУБД будет автоматически выполнять при каждом возникновении события.

Триггер DML — это предварительно определенное действие или последовательность действий, автоматически производимых при выполнении запросов обновления, добавления или удаления данных к определенной таблице (представлению) БД.

Триггер создается для заданной таблицы (представления) и сохраняется в БД как часть метаданных. Он является мощным инструментом контроля изменений данных в БД, а также помогает автоматизировать операции, которые должны выполняться в этом случае. Триггер выполняется после проверки правил обновления данных.

Триггер включает в себя два компонента:

- 1) ограничения, для реализации которых он создается;
- 2) событие, характеризующее возникновение ситуации, требующей проверки ограничений.

В каждой СУБД структура триггера включает в себя следующие основные секции:

- 1) секция определения имени триггера;
- 2) секция, показывающая, для какой таблицы (представления) создается триггер;
- 3) секция, показывающая, для какого события или событий выполнять триггер (INSERT — при создании строки, DELETE — при удалении строки и UPDATE — при изменении);
- 4) тело триггера, содержащее операторы процедурного SQL и реализующее требуемые ограничения или бизнес-правила.

Предусмотренное действие производится за счет выполнения определенной операции или последовательности операций, с помощью которых реализуется логика, требуемая для удовлетворения ограничений.

События возникают при изменении содержимого таблицы (представления). Действие, вызываемое событием, задается последовательностью запросов SQL и операторов процедурного языка СУБД.

Триггеры являются частью работы транзакции, в которой событие DML изменяет состояние строки. Если транзакция успешно подтверждается, то все действия триггеров принимаются. Если будет выполнен откат транзакции, то все действия триггера будут отменены.

Рассмотрим особенности работы триггеров DML в СУБД Firebird.

Для БД Firebird триггер может быть создан в IBExpert после подключения к базе:

- с помощью конструктора, вызываемого по Ctrl + N на закладке **Триггеры** инспектора объектов БД;
- из пункта **Новый триггер** главного меню **База данных**;
- выполнением запроса в SQL-редакторе или в скрипте.

В Firebird триггер определяется запросом CREATE TRIGGER, имеющим формат

```
CREATE TRIGGER имя_триггера
  FOR { базовая_таблица | представление }
  [ACTIVE | INACTIVE]
  { BEFORE | AFTER } <событие1> [OR <событие2> [OR <событие3>] ]
  [POSITION приоритет_триггера]
AS <тело_триггера>
```

где <событие> ::= { DELETE | INSERT | UPDATE }

Определение триггера состоит из заголовка и тела. Заголовок содержит:

- имя создаваемого триггера;
- имя таблицы или модифицируемого представления, для которых создается триггер;
- состояние триггера (активный или неактивный);
- описание связи с событиями, при наступлении которых триггер должен сработать;
- приоритет выполнения триггера над другими триггерами, если те связаны с той же таблицей (представлением).

В заголовке триггера его активность определяется указанием ключевого слова ACTIVE. Триггер можно «отключить», если использовать в заголовке триггера ключевое слово INACTIVE. По умолчанию любой триггер создается активным, т. е. не обязательно указывать ключевое слово ACTIVE при определении нового триггера. Явное указание ACTIVE может потребоваться при «включении» неактивного триггера (как можно модифицировать ранее созданный триггер, будет подробно описано в подпараграфе 6.6.1.3).

Определяя триггер, можно установить момент срабатывания триггера. В Firebird триггер может выполняться в одной из двух фаз: BEFORE или AFTER. BEFORE указывает на то, что триггер должен сработать до указанных событий, а AFTER активизирует работу триггера после наступления указанных событий.

Триггеры BEFORE чаще всего используется в следующих ситуациях:

- когда действие триггера должно определить, стоит ли выполняться команде триггера. Используя триггер BEFORE, можно избежать ненужной обработки команды триггера;
- для автоматического вычисления значений столбцов перед тем, как выполнить команду триггера INSERT или UPDATE. Триггеры BEFORE позволяют принудительно установить значения определенных столбцов, базируясь на значениях других столбцов в той же самой строке, которые поставляются командой триггера INSERT или UPDATE.

Триггеры AFTER используются, если необходимо, чтобы команда триггера завершилась полностью перед выполнением действия триггера.

Если существует триггер BEFORE, то триггер AFTER может также присутствовать и выполнять другие действия для той же команды триггера.

DELETE, INSERT и UPDATE задают три типа событий, на которые триггер «реагирует» соответственно при удалении, вставке или обновлении таблицы, для которой создан триггер. В любом случае срабатывает триггер в Firebird для каждой отдельной строки, затрагиваемой событием.

Возможно объединение действий для двух или трех событий DML в одном триггере. В то же время в БД может быть создано несколько триггеров, которые ассоциированы с одной и той же таблицей, одним и тем же событием и имеют одну и ту же фазу. При этом для одной таблицы (представления) можно создать не более 32 768 триггеров. Порядок выполнения таких триггеров устанавливается с помощью параметра секции POSITION.

Значение приоритет_триггера может меняться от 0 до 32 767. Триггеры с меньшими номерами выполняются первыми. Если же несколько триггеров имеют один и тот же приоритет, то они выполняются в алфавитном порядке их имен.

Тело триггера задает последовательность действий, которая будет реализована СУБД при наступлении контролируемых событий над заданной таблицей или представлением.

Тело триггера, как и тело ХП, состоит из списка объявления используемых локальных переменных и блока выполняемых операторов:

```
<тело_триггера> ::= [<список_объявления>]  
                   <блок_операторов>
```

где <список_объявления> и <блок_операторов> имеют тот же синтаксис, который был описан в подпараграфе 6.2.1 при рассмотрении ХП.

Microsoft SQL Server позволяет создать DML-триггер запросом следующего формата:

```
CREATE TRIGGER имя_триггера
ON { базовая_таблица | представление }
[ WITH <опция_триггера> [ ,... ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS { <тело триггера> | EXTERNAL NAME <метод> }
```

где

```
<опция_триггера> ::= { ENCRYPTION | EXECUTE AS <контекст> }
<метод> ::= имя_сборки.имя_класса.имя_метода
```

Имя триггера не может превышать 128 символов и должно быть уникальным в схеме БД.

В MS SQL по умолчанию триггер включен, но его можно отключить командой вида

```
DISABLE TRIGGER { <имя_триггера> [ ,... ] | ALL }
ON { <имя_объекта> | DATABASE | ALL SERVER } [ ; ]
```

В необязательной секции WITH могут быть заданы одна или две опции триггера. Опция ENCRYPTION указывает, что текст триггера должен кодироваться при помещении в БД. Опция EXECUTE AS определяет контекст безопасности обращения к триггеру.

Триггер может выполняться в различных фазах: AFTER, FOR, INSTEAD OF. Ключевые слова FOR и AFTER используются как синонимы и означают, что к триггеру происходит обращение после выполнения изменений БД и после проверки соответствия изменяемых, удаляемых или добавляемых данных декларативной целостности данных в базе. Такой триггер сам может рекурсивно вызвать себя, если в нем применяется операция, вызвавшая этот триггер. Однако глубина вложенности не может превышать числа 32, как и в случае с ХП. Триггеры AFTER не могут быть определены на представлениях. Если триггер создан как INSTEAD OF, то он подменяет вызвавшее его событие: указанные действия в самом операторе, вызвавшем неявно триггер, не выполняются, а будут выполнены только действия триггера. Следует отметить, что если в триггере INSTEAD OF выполняется та же самая операция, по которой был запущен на выполнение этот триггер, то дополнительный вызов триггера не произойдет.

Для одной операции в одной таблице может быть задано несколько триггеров INSTEAD OF и несколько триггеров AFTER. Сначала запускаются триггеры INSTEAD OF, а затем триггеры AFTER.

Триггеры могут быть созданы для реагирования на операцию DML или группу операций DML. Если в операции добавления, изменения или удаления задействовано более одной строки, то триггеры будут вызываться не для каждой строки, а для всех включенных в операцию строк.

Опция WITH APPEND позволяет создавать несколько триггеров каждого типа.

Порядок запуска триггеров одинаковых типов, созданных для одной и той же таблицы, в MS SQL Server никак не определен и не гарантирован. Однако можно изменять порядок запуска триггеров с помощью системной процедуры SP_SETTRIGGERORDER [7]. Процедура позволяет указать, какие триггеры AFTER срабатывают первыми, а какие — последними. Триггеры AFTER, срабатывающие между первым и последним триггерами, выполняются в неопределенном порядке.

При создании триггера с опцией NOT FOR REPLICATION запрещается его запуск во время выполнения изменения таблиц механизмами репликации.

Тело триггера определяет набор SQL-операторов и команд, которые будут выполнены при запуске триггера. При этом внутри триггера не допускается выполнение ряда операций, таких, например, как:

- создание, изменение и удаление БД;
- восстановление резервной копии БД или журнала транзакций.

Выполнение этих команд не разрешено, так как они не могут быть отменены в случае отката транзакции, в которой выполняется триггер.

Вместо тела триггера при его определении в секции EXTERNAL NAME может быть указан метод управляемого кода.

В СУБД Oracle определение DML-триггеров поддерживается в формате

```
CREATE [ OR REPLACE ] TRIGGER имя_триггера
{ BEFORE | AFTER | INSTEAD OF }
<событие1> [OR <событие2> [OR <событие3>] ]
ON { базовая_таблица | представление |
    NESTED TABLE <список_столбцов> OF представление }
[ REFERENCING { OLD AS <значение> | NEW AS <значение> | PARENT
AS <значение> } ]
[ FOR EACH ROW ]
[ WHEN (<условие>) { <тело_блока> | CALL имя_процедуры (<список_
параметров>) } ]
<тело триггера>
```

где

```
<событие> ::= { DELETE | INSERT | UPDATE [ OF <список_столбцов> ] }
```

Триггер DML создается для базовой таблицы или представления. При этом триггер `INSTEAD OF`, который выполняется вместо события DML, в Oracle может быть создан только для представления. В таком случае используется секция `NESTED TABLE`, которая не поддерживается при определении `BEFORE` или `AFTER`-триггеров.

Триггер может срабатывать при вставке, обновлении или удалении строк. При операции `MERGE` срабатывают триггеры на изменение, вставку или удаление строк в зависимости от операции со строкой. Если триггер реагирует на событие `UPDATE`, то может быть указан список столбцов, на обновление которых будет реагировать триггер. Если список столбцов не указан, то триггер реагирует при обновлении любого столбца ассоциированной таблицы.

Секция `REFERENCING` может специфицироваться в теле триггера строк для того, чтобы избежать конфликтов между корреляционными именами и именами таблиц в случае, если таблица имеет имя «`OLD`» или «`NEW`». Такая ситуация редкая, и эта опция почти никогда не применяется.

Триггер может запускаться один раз для каждой задействованной строки или один раз за все время выполнения команды (независимо от числа затронутых строк) [1]. Триггер, срабатывающий для каждой строки, называется *триггером событий на уровне строки* (*row-level trigger*), а триггер, срабатывающий один раз на все событие DML, называется *триггером событий на уровне операции* (*statement-level trigger*). Опция `FOR EACH ROW` определяет, является ли триггер триггером выражения или триггером строки. Если опция включена, то триггер реагирует отдельно для каждой строки таблицы.

Если в триггере уровня строки попытаться получить или изменить данные целевой таблицы, то Oracle не позволит это сделать и выдаст ошибку `ORA-04091` вида «`table ... is mutating, trigger/function may not see it`» (таблица ... мутировала, триггер/функция не может ее видеть) [1, 24]. *Мутировавшей таблицей* в Oracle называется таблица, модифицированная запросом DML, который инициировал событие. Операторы в теле триггера не могут считать или модифицировать таблицу, затронутую инициировавшим запуск триггера запросом. Для обхода данной проблемы можно:

- использовать триггеры уровня операции;
- применить автономную транзакцию в триггере;
- использовать сторонние структуры (коллекции уровня пакета);
- применить составной триггер;
- изменить алгоритм решения с выносом функционала из триггера.

Следует учесть, что если в Oracle для таблицы существует несколько триггеров одного типа, СУБД выполняет их в произвольном

порядке. Если порядок срабатывания триггеров имеет значение, по возможности их следует объединить в один триггер.

Секция WHEN может быть задана при создании триггера для выполнения определенного PL/SQL-блока или вызова ХП при истинности некоторого условия. Условие не может содержать запрос, вызов PL/SQL функций и методов. Секция может быть определена только для триггеров уровня строки, не может быть определена для INSTEAD OF-триггеров.

Тело триггера в Oracle может содержать запросы SELECT (SELECT ... INTO или SELECT в определениях курсоров), любые запросы DML. В теле триггера не допускаются предложения DDL. Нельзя также управлять транзакциями в контексте триггера, поэтому внутри тела триггера не допускается использование операторов ROLLBACK, COMMIT и SAVEPOINT. Процедуры, вызываемые из триггера, также не могут выполнять перечисленные выше операторы управления транзакциями, так как исполняются внутри контекста тела триггера.

Начиная с версии Oracle Database 11g, поддерживаются составные (COMPOUND) триггеры, которые включают в одном блоке обработку всех видов DML триггеров [18, 24]. Срабатывают такие триггеры при разных событиях и в разные моменты времени, на уровне запроса или строки, при вставке/обновлении/удалении, до или после события.

Для создания составного триггера используется запрос формата

```
CREATE [ OR REPLACE ] TRIGGER имя_триггера
FOR <событие1> [OR <событие2> [OR <событие3>] ]
ON { базовая_таблица | представление }
[ REFERENCING { OLD AS <значение> | NEW AS <значение> | PARENT
AS <значение> } ]
COMPOUND TRIGGER
[<список_объявления>]
<выполняемая секция1>
[<выполняемая секция2>...]
```

где

```
<выполняемая секция> ::= секция IS BEGIN <блок_операторов> END секция;
секция ::= { BEFORE STATEMENT | BEFORE EACH ROW |
            AFTER STATEMENT | AFTER EACH ROW |
            INSTEAD OF EACH ROW }
```

Составной триггер может содержать следующие секции:

- BEFORE STATEMENT — выполняется один раз перед выполнением запроса DML;
- AFTER STATEMENT — выполняется один раз после выполнения запроса DML;

- BEFORE EACH ROW — выполняется один раз для каждой строки перед самым действием;
- AFTER EACH ROW — выполняется один раз для каждой строки после действия;
- INSTEAD OF EACH ROW— выполняется один раз для каждой строки вместо действия.

Таким образом, код составного триггера монолитен, но каждая секция выполняется в разное время. Если в триггере нет ни секции BEFORE STATEMENT, секции ни AFTER STATEMENT, и запрос DML не затрагивает ни одну строку, такой триггер не срабатывает.

При работе с составными триггерами существует ряд ограничений:

- нельзя обращаться к псевдозаписям OLD, NEW или PARENT в секциях уровня выражения (BEFORE STATEMENT и AFTER STATEMENT);
- изменять значения полей псевдозаписи NEW можно только в секции BEFORE EACH ROW;
- исключения, сгенерированные в одной секции, нельзя обрабатывать в другой секции;
- если используется оператор GOTO, он должен указывать на код в той же секции.

Рассмотрим пример триггера для одного события. Пусть бизнес-правила требуют перед удалением строк в таблице Executor сохранять в таблице History информацию о ФИО всех исполнителей ремонтных заявок и дате, до которой данная информация была актуальна. Скрипт для создания триггера Executor_Delete, реализующего это бизнес-правило, в Firebird будет выглядеть следующим образом:

```

/* создание таблицы History */
CREATE TABLE History
  (Id INTEGER PRIMARY KEY, Executor_List VARCHAR(300),
   FixDate DATE);
/* создание генератора */
CREATE SEQUENCE Gen_History;
/* установка начального значения генератора */
ALTER SEQUENCE Gen_History;
SET TERM !! ; /* установка нового терминатора */
/* определение триггера */
CREATE TRIGGER Executor_Delete FOR Executor ACTIVE BEFORE DELETE
AS
  DECLARE VARIABLE Line VARCHAR(300);
  DECLARE VARIABLE OneLine VARCHAR(20);
BEGIN
  /* Запись ФИО исполнителей в переменную Line */
  Line = '';
  FOR
    SELECT Fio FROM Executor INTO :OneLine

```

```

DO
    IF (OneLine IS NOT NULL) THEN Line = Line || :OneLine || ', ';
    /* вставка строки в таблицу History */
    INSERT INTO History VALUES(
        NEXT VALUE FOR Gen_History, :Line, CURRENT_DATE);
END!! /* конец CREATE TRIGGER */
SET TERM ; !! /* установка стандартного терминатора */

```

В приведенном скрипте сначала производится создание таблицы History, а также определение и установка начального нулевого значения генератора Gen_History, используемого для получения уникального значения столбца первичного ключа таблицы History. Затем создается триггер Executor_Delete, который запускается перед наступлением события удаления строки из таблицы Executor. Триггер осуществляет запись в локальную переменную Line ФИО всех исполнителей, разделенных запятыми. Полученное значение переменной Line и дата, до которой информация была актуальна (дата удаления строк, т. е. текущая дата), записываются в таблицу History с помощью запроса INSERT, размещенного в теле триггера.

В Oracle код определения триггера будет выглядеть так:

```

/* создание таблицы History */
CREATE TABLE History
    (Id INTEGER PRIMARY KEY, Executor_List VARCHAR2(300),
    FixDate DATE);
/* создание генератора и установка начального значения */
CREATE SEQUENCE Gen_History START WITH 1 MINVALUE 1;
/* определение триггера */
CREATE TRIGGER Executor_Delete BEFORE DELETE ON Executor
DECLARE
    Line VARCHAR2(300);
    CURSOR FioCursor IS
        SELECT Fio FROM Executor;
BEGIN
    /* Запись ФИО исполнителей в переменную Line */
    Line := '';
    FOR i IN FioCursor LOOP
        Line := Line || i.Fio || ', ';
    END LOOP;
    /* вставка строки в таблицу History */
    INSERT INTO History VALUES(
        Gen_History.NEXTVAL, Line, CURRENT_DATE);
END;
/

```

Как можно заметить, начальное значение генератора в Oracle устанавливается сразу при его создании, затем изменить его нельзя. Триггер объявляется как BEFORE DELETE и работает, используя курсор FioCursor.

Microsoft SQL Server не поддерживает BEFORE-триггеры, и задача выполнения триггером каких-либо действий до возникновения события решается несколько иначе, чем в двух предыдущих СУБД. Можно реализовать триггер как в виде FOR-триггера, так и в виде INSTEAD OF-триггера, используя при этом логические (концептуальные) таблицы.

Возможность использования логических таблиц либо ряда контекстных переменных, которые нельзя применять в ХП, является характерной особенностью триггеров в любой СУБД.

В MS SQL Server возможно обращение к так называемым логическим или концептуальным таблицам DELETED и INSERTED в триггерах DML в формате

```
{DELETED | INSERTED}.столбец
```

К таблице DELETED может обращаться триггер, запускаемый при удалении строк таблицы [8]. Эта логическая таблица содержит все строки, удаляемые вызвавшим триггер запросом. Похожим образом таблица INSERTED содержит все добавляемые строки в основную таблицу. Все они являются временными и располагаются в оперативной памяти. В зависимости от вида соответствующего оператора логические таблицы могут содержать более одной строки. Следует обратить внимание на отсутствие логической таблицы с именем, например, updated, которая содержала бы данные, измененные запросом UPDATE. Все необходимые данные находятся в таблицах DELETED и INSERTED: таблица DELETED содержит данные до их изменения оператором, а таблица INSERTED — после их изменения.

Предыдущую задачу по созданию триггера, сохраняющего историю об исполнителях заявок перед удалением, в MS SQL Server можно решить с помощью, например, AFTER-триггера. Создание срабатывающего после наступления события AFTER-триггера — наиболее распространенный подход в MS SQL Server в ситуации, когда требуется создать триггер, выполняющий действия до возникновения события. Скрипт будет таким:

```
CREATE TABLE History
  (Id INTEGER PRIMARY KEY, Executor_List VARCHAR(300),
  FixDate DATE);
GO
/* создание генератора */
CREATE SEQUENCE Gen_History;
GO
/* установка начального значения генератора */
ALTER SEQUENCE Gen_History RESTART WITH 0;
GO
/* определение триггера */
CREATE TRIGGER Executor_Delete ON Executor FOR DELETE
```

```

AS
DECLARE @Line VARCHAR(300) = '';
DECLARE @OneLine VARCHAR(20) = '';
DECLARE curs CURSOR
FOR
    SELECT Fio FROM DELETED
    UNION ALL
    SELECT Fio FROM Executor;
BEGIN
    IF EXISTS(SELECT * FROM DELETED)
    BEGIN
        SET NOCOUNT ON;
        OPEN curs;
        FETCH NEXT FROM curs INTO @OneLine;
        WHILE @@FETCH_STATUS = 0
        BEGIN
            SET @Line = @Line + @OneLine + ', ';
            FETCH NEXT FROM curs INTO @OneLine;
        END;
        CLOSE curs;
        /* вставка строки в таблицу History */
        INSERT INTO History VALUES(NEXT VALUE FOR Gen_History,
                                   @Line, GETDATE());

    END;
    DEALLOCATE curs;
END /* конец CREATE TRIGGER */
GO

```

Триггер `Executor_Delete` в MS SQL Server первым делом выполняет проверку наличия удаляемых строк в виртуальной таблице `DELETED`. Если таблица `DELETED` не пустая, то далее с помощью объявленного явного курсора `Curs` выполняется последовательная обработка ее строк, а также строк таблицы `Executors`. Информация сохраняется в таблице `History`. Триггер выполняется в рамках неявной транзакции запроса `Delete`, поэтому, если удаление не удастся, действия в триггере также не будут выполнены.

Реализовать предыдущий триггер можно и как `INSTEAD OF`. В таком случае он будет подменять собой событие удаления, поэтому после выполнения любых действий в таком триггере должно быть запросом выполнено удаление данных из таблицы, для которой создан триггер. Запрос для определения триггера:

```

CREATE TRIGGER Executor_Delete ON Executor INSTEAD OF DELETE
AS
DECLARE @Line VARCHAR(300) = '';
DECLARE @OneLine VARCHAR(20) = '';
DECLARE curs CURSOR
FOR
    SELECT Fio FROM Executor;
BEGIN
    IF EXISTS(SELECT * FROM DELETED)

```

```

BEGIN
  SET NOCOUNT ON;
  OPEN curs;
  FETCH NEXT FROM curs INTO @OneLine;
  WHILE @@FETCH_STATUS = 0
  BEGIN
    SET @Line = @Line + @OneLine + ', ';
    FETCH NEXT FROM curs INTO @OneLine;
  END;
  CLOSE curs;
  /* вставка строки в таблицу History */
  INSERT INTO History VALUES(
    NEXT VALUE FOR Gen_History, @Line, GETDATE());
  /* удаление строк целевой таблицы */
  DELETE Executor
  FROM DELETED D
  WHERE Executor.ExecutorCD = D.ExecutorCD;
END;
DEALLOCATE curs;
END /* конец CREATE TRIGGER */
GO

```

Приведенный триггер, в отличие от предыдущего, использует курсор, обрабатывающий строки только таблицы Executor. Поскольку триггер подменяет собой событие удаления, то все строки на момент срабатывания триггера еще доступны в исходной таблице. После основных действий по сохранению данных в таблице History выполняется удаление строк из целевой таблицы Executor. Строки удаляются по ключу ExecutorCD, найденному среди строк таблицы DELETED, которая заполнена при срабатывании INSTEAD OF-триггера.

В Firebird основным средством реализации возможностей триггеров по отслеживанию целостности данных являются контекстные переменные OLD и NEW. Переменные используются в таком виде:

{OLD | NEW}.столбец

Контекстная переменная OLD ссылается в строке на *текущие* или *предыдущие* значения, обновляемые или удаляемые соответственно запросами UPDATE или DELETE в таблице, для которой создан триггер. Таким образом, переменная OLD не используется при вставке, так как старого значения не существует.

Переменная OLD является контекстной переменной только для чтения, поэтому попытка присвоить какое-либо значение переменной OLD.столбец будет отклонена.

Контекстная переменная NEW ссылается в строке на *новые значения*, которые будут вставлены или обновлены соответственно запросами INSERT или UPDATE для таблицы, для которой создан триггер.

гер. Переменная NEW не используется при удалении, так как новое значение не создается.

Переменная NEW является контекстной переменной только для чтения в триггерах, которые срабатывают после наступления событий (фаза триггера — AFTER).

Эти переменные могут быть использованы, в частности, для:

- реализации получения автоинкрементных значений;
- получения значений по умолчанию;
- проверки входных данных, и при необходимости, их изменения;
- получения значений столбцов для модификации других столбцов.

Контекстные переменные OLD и NEW могут использоваться в триггерах для нескольких событий. Но при этом могут возникнуть ситуации, когда для одного события, на которое реагирует триггер, контекстная переменная может использоваться, а для другого — нет. В этом случае ссылка на NEW.столбец в контексте удаления или на OLD.столбец в контексте добавления ошибкой не является — просто будет возвращен NULL.

В Oracle поддерживаются так называемые псевдозаписи OLD и NEW, позволяющие обратиться к полям изменяемой записи и получить значения полей до изменения и значения полей после изменения. Данные псевдозаписи доступны в виде OLD.столбец, NEW.столбец при использовании в условии секции WHEN либо с двоеточием :OLD.столбец, :NEW.столбец во всех остальных случаях. С помощью секции REFERENCING можно изменять имена записей. Следует учесть, что псевдозаписи доступны для использования только у триггеров строк или у составных (compound) триггеров с секциями уровня записи.

Простейший триггер для автоматического формирования значения первичного ключа, например таблицы Street, в Firebird можно определить так:

```
CREATE TRIGGER Street_Cd FOR Street
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
    IF (NEW.StreetCD IS NULL) THEN
        NEW.StreetCD = NEXT VALUE FOR Gen_Street;
END
```

Здесь контекстная переменная NEW используется в качестве имени таблицы (Street), в которую вставляется строка. Если вставка очередной строки в таблицу Street производится запросом INSERT без указания значения первичного ключа, то это значение будет генерироваться автоматически.

При нарушении ограничения триггер должен формировать информативное сообщение об исключительной ситуации. Рассмотрим пример создания триггера, контролирующего вставку строк в таблицу Request.

Допустим, существует такое бизнес-ограничение, по которому каждый исполнитель может быть назначен на выполнение не более трех ремонтных заявок в день. При попытке зарегистрировать для исполнителя «лишнюю» ремонтную заявку должно выдаваться сообщение. Скрипт создания триггера для решения данной задачи в Firebird:

```
SET TERM !! ; -- определение нового терминатора
/* определение триггера */
CREATE TRIGGER Num_Request FOR Request
  ACTIVE BEFORE INSERT POSITION 0
AS
DECLARE ExecKod SMALLINT;
DECLARE God DATE;
DECLARE NumRows SMALLINT;
BEGIN
  FOR
    SELECT ExecutorCD, IncomingDate, COUNT(*)
    FROM Request WHERE ExecutionDate IS NULL
    GROUP BY 1,2
    INTO :ExecKod, :God, :NumRows
  DO
    BEGIN
      IF (ExecKod = New.ExecutorCD
        AND God = NEW.IncomingDate
        AND NumRows >= 3)
        /* генерация исключительной ситуации, если для исполнителя в день
        уже зарегистрировано более 3 заявок */
        THEN Exception
          Ins_Restrict 'Превышен лимит заявок для исполнителя в день';
    END
  END !!
  -- конец CREATE TRIGGER
SET TERM ; !! /* установка стандартного терминатора */
```

Созданный триггер Num_Request при попытке назначить исполнителю на выполнение более трех заявок в день вызывает исключение Ins_Restrict, определенное при создании учебной БД, и предотвращает вставку строки в таблицу Request.

В MS SQL Server аналогичный триггер может быть создан так:

```
CREATE TRIGGER Num_Request ON Request AFTER INSERT
AS
BEGIN
  IF EXISTS
    (SELECT *
     FROM Request R INNER JOIN Inserted I
```

```

        ON R.ExecutorCD = I.ExecutorCD
        AND R.IncomingDate = I.IncomingDate
WHERE R.ExecutionDate IS NULL
GROUP BY R.ExecutorCD,R.IncomingDate
HAVING COUNT(*) >= 3)
BEGIN
    RAISERROR ('Превышен лимит заявок для исполнителя в день',
        16, 1);
    ROLLBACK TRANSACTION;
    RETURN;
END;
END;
GO

```

Данный триггер срабатывает после вставки строк и использует виртуальную таблицу INSERTED для анализа добавляемых строк. При попытке назначить исполнителю на выполнение более трех заявок в день вызывается исключение с помощью оператора RAISERROR, и транзакция вставки отменяется.

В СУБД Oracle можно реализовать аналогичный триггер уровня строк:

```

CREATE TRIGGER NUM_REQUEST
BEFORE INSERT
ON REQUEST
FOR EACH ROW
DECLARE
    ExecKod      SMALLINT;
    God          DATE;
    NumRows      SMALLINT;
    ErrorMessage VARCHAR2(255);
    Ins_Restrict EXCEPTION;
BEGIN
    FOR Rec IN (SELECT ExecutorCD AS ExecKod,
                    INCOMINGDATE AS God,
                    COUNT(*) AS NumRows
                FROM Request
                WHERE ExecutionDate IS NULL
                GROUP BY ExecutorCD,
                    IncomingDate)
    LOOP
        IF (Rec.ExecKod = :NEW.ExecutorCD
            AND Rec.God = :NEW.IncomingDate
            AND Rec.NumRows >= 3)
        THEN
            RAISE Ins_Restrict;
        END IF;
    END LOOP;
EXCEPTION
    WHEN Ins_Restrict THEN
        ErrorMessage := 'Превышен лимит заявок для исполнителя в день';
END;
/

```

Приведенный пример триггера Oracle демонстрирует не только использование псевдозаписи NEW, но также объявление и вызов пользовательского ограничения (Ins_Restrict) в коде PL/SQL.

В некоторых СУБД в триггерах для нескольких событий с целью поддержания условных переходов между событиями используются логические контекстные переменные. Например, в триггерах Firebird и Oracle поддерживаются переменные UPDATING, INSERTING, DELETING. Они употребляются для задания определенных действий в зависимости от типа события DML и доступны в качестве предикатов в условных структурах, выполняющих операции изменения состояния данных. Например в теле триггера Firebird:

```
IF (INSERTING OR DELETING) THEN
    NEW.ID = GEN_ID(G_GENERATOR_1, 1);
```

Рассмотрим пример триггера для нескольких событий, реализующего бизнес-правило вида активатора операций. Пусть требуется в некоторой таблице Journal фиксировать имена пользователей, которые добавляют, обновляют или удаляют данные в таблице NachislSumma, идентификатор факта начисления (NachislFactCD), признак типа события ('INS' для вставки, 'EDIT' для обновления и 'DEL' для удаления), а также дату и время совершения операции. Скрипт для создания в Firebird такого триггера с именем Usr_Action:

```
/* создание таблицы Journal */
CREATE TABLE Journal (Usr_Name VARCHAR(20),
                       Nach_ID INTEGER, DML_Event VARCHAR(4),
                       Date_Time TIMESTAMP);
SET TERM !! ; /* установка нового терминатора */
/* определение триггера */
CREATE TRIGGER Usr_Action FOR NachislSumma
  ACTIVE AFTER INSERT OR UPDATE OR DELETE
AS
DECLARE VARIABLE Nach INTEGER;
DECLARE VARIABLE Event VARCHAR(4);
BEGIN
  IF (DELETING) THEN
    BEGIN
      Nach = OLD.NachislFactCD;
      Event = 'DEL';
    END
  ELSE
    BEGIN
      Nach = NEW.NachislFactCD;
      IF (UPDATING) THEN
        Event = 'EDIT';
      ELSE
        Event = 'INS';
      END
    END
  /* вставка строки в таблицу Journal */
```

```
INSERT INTO Journal VALUES
(USER, :Nach, :Event, CURRENT_TIMESTAMP);
END!! /* конец CREATE TRIGGER */
SET TERM ; !! /* установка стандартного терминатора */
```

Чтобы проверить действие созданного триггера `Usr_Action`, от имени пользователя `SYSDBA` выполним последовательно три запроса:

```
INSERT INTO NachislSumma VALUES (80, '005488', 1, 63, 1, 2016);
UPDATE NachislSumma SET ServiceCD = 2 WHERE NachislFactCD = 80;
DELETE FROM NachislSumma WHERE NachislFactCD = 80;
```

Данные, полученные в результате выполнения запроса

```
SELECT * FROM Journal;
```

и помещенные в таблицу `Journal`, представлены на рис. 6.17.

USR_NAME	NACH_ID	DML_EVENT	DATE _TIME
SYSDBA	80	INS	12.05.2018 10:02
SYSDBA	80	EDIT	12.05.2018 10:03
SYSDBA	80	DEL	12.05.2018 10:04

Рис. 6.17. Данные таблицы `Journal`

В Oracle задачу можно реализовать путем создания триггера для нескольких событий следующим скриптом:

```
CREATE TABLE Journal (Usr_Name VARCHAR2(20), Nach_ID INTEGER,
DML_Event VARCHAR2(4), Date _Time DATE);
CREATE TRIGGER Usr_Action
AFTER INSERT OR UPDATE OR DELETE ON NachislSumma FOR EACH ROW
DECLARE
    Nach INTEGER;
    Event VARCHAR2(4);
BEGIN
    IF DELETING THEN
        Nach := :OLD.NachislFactCD;
        Event := 'DEL';
    ELSE
        Nach := :NEW.NachislFactCD;
        IF UPDATING THEN
            Event := 'EDIT';
        ELSE
            Event := 'INS';
        END IF;
    END IF;
    INSERT INTO Journal VALUES (USER, Nach, Event, SYSDATE);
```



```
END;  
/
```

Переменные, позволяющие определить тип события, не поддерживаются в триггерах MS SQL Server. Определить тип события можно косвенно, проанализировав количество строк в таблицах INSERTED и DELETED: если количество строк в них совпадает, то это операция UPDATE, если в INSERTED нет строк, то это DELETE, если в DELETED нет строк, — INSERT. Однако если требуется выполнять принципиально различные действия при наступлении различных типов событий в MS SQL Server, то рекомендуется создавать отдельные триггеры для каждого типа события.

Описание всех имеющихся в БД триггеров (как системных, так и пользовательских) обычно можно получить по информации системных таблиц СУБД. Например, в Firebird информация о триггерах хранится в системной таблице RDB\$TRIGGERS. При использовании утилиты IBExpert все пользовательские триггеры отображаются в окне инспектора объектов БД. В MS SQL Server информацию о триггерах можно получить из системной таблицы SYSOBJECTS, в Oracle информацию обо всех триггерах схемы можно получить из словаря данных — представления USER_TRIGGERS.

6.6.1.2. Примеры триггеров

1. Создадим триггер Td_Executor, который при удалении строки из таблицы Executor сначала удаляет все связанные с ней строки из таблицы Request, а затем удаляет саму строку из таблицы Executor, тем самым обеспечивая целостность данных. В Firebird такой триггер можно создать так:

```
CREATE TRIGGER Td_Executor FOR Executor AFTER DELETE AS  
BEGIN  
    DELETE FROM Request WHERE ExecutorCD = OLD.ExecutorCD;  
    DELETE FROM Executor WHERE ExecutorCD = OLD.ExecutorCD;  
END;
```

или так:

```
CREATE TRIGGER Td_Executor FOR Executor BEFORE DELETE AS  
BEGIN  
    IF (EXISTS (SELECT *  
                FROM Request  
                WHERE ExecutorCD = OLD.ExecutorCD))  
    THEN  
        DELETE FROM Request WHERE ExecutorCD = OLD.ExecutorCD;  
END;
```

Во втором случае в коде триггера осуществляется проверка наличия у подлежащей удалению строки, идентифицируемой значением

первичного ключа OLD.ExecutorCD, подчиненных строк. Если такие обнаруживаются, то к ним применяется запрос DELETE. Таким образом, дочерние строки исключаются из таблицы до того, как это произойдет с родительской строкой.

В MS SQL Server запрос определения аналогичного триггера:

```
CREATE TRIGGER Td_Executor ON Executor AFTER DELETE AS
BEGIN
    DELETE FROM Request
        WHERE ExecutorCD IN (SELECT D.EXECUTORCD FROM DELETED D);
    DELETE FROM Executor
        WHERE ExecutorCD IN (SELECT D.EXECUTORCD FROM DELETED D);
END;
```

В Oracle это будет триггер уровня строк:

```
CREATE TRIGGER Td_Executor
AFTER DELETE ON Executor FOR EACH ROW
BEGIN
    DELETE FROM Request WHERE ExecutorCD = :OLD.ExecutorCD;
    DELETE FROM Executor WHERE ExecutorCD = :OLD.ExecutorCD;
END;
```

2. Пусть необходимо разработать SQL-сценарий по созданию триггера, предотвращающего вставку NULL значений в столбцы внешних ключей AccountCD, ExecutorCD и FailureCD, а также в столбцы IncomingDate и Executed таблицы Request. Кроме того, дата выполнения заявки не должна быть ранее даты ее регистрации. Если значения, вводимые в перечисленные столбцы, удовлетворяют требованиям, то необходимо вставить в столбец первичного ключа RequestCD очередное значение, полученное с помощью генератора. При попытке вставить недопустимые значения должен сработать триггер и выдать сообщение о соответствующей исключительной ситуации, т. е. триггер должен запускаться при каждой вставке строки в таблицу Request. Ниже приведен текст SQL-сценария, реализующего поставленную задачу в Firebird:

```
CREATE EXCEPTION E_Request ' Insert null value into Request ';
CREATE EXCEPTION E_I_Request ' ExcecutionDate < IncomingDate ';
SET TERM !! ; -- определение нового терминатора
CREATE TRIGGER Request_Insert FOR Request BEFORE INSERT
AS
/* конец заголовка и начало тела триггера */
BEGIN
    IF ((NEW.AccountCD IS NULL) OR (NEW.ExecutorCD IS NULL) OR
        (NEW.FailureCD IS NULL) OR (NEW.IncomingDate IS NULL) OR
        (NEW.Executed IS NULL))
    THEN
        /* генерация исключительной ситуации, если значения столбцов
```

```

    AccountCD, ExecutorCD, FailureCD или IncomingDate
    содержат NULL */
    EXCEPTION E_Request;
ELSE
    /* проверка значений вставляемой строки в таблицу Request;
    при этом переменная NEW представляет собой ссылку на
    вставляемое значение в указанный столбец внешнего ключа
    таблицы Request */
    IF (NEW.ExecutionDate < NEW.IncomingDate)
    THEN
    /* генерация исключительной ситуации, если
    ExecutionDate < IncomingDate */
        EXCEPTION E_I_Request;
    ELSE
    /* присвоение столбцу первичного ключа таблицы Request
    очередного значения генератора Gen_Request */
        BEGIN
            NEW.RequestCD = NEXT VALUE FOR Gen_Request;
        END
    END !!          -- конец CREATE TRIGGER
    /* установка стандартного терминатора */
    SET TERM ; !!

```

Следует обратить внимание на способ присвоения полю RequestCD (первичный ключ таблицы Request) очередного значения: здесь также контекстная переменная NEW используется в качестве имени таблицы (Request), в которую вставляется строка. В данном случае нельзя вставлять внутри тела триггера значение первичного ключа в таблицу Request с помощью запроса INSERT (этот триггер создан именно для события INSERT таблицы Request, он фактически будет пытаться вызвать сам себя). Таким образом, вставка очередной строки в таблицу Request должна производиться запросом INSERT без указания значения первичного ключа, которое будет генерироваться автоматически:

```

INSERT INTO Request
    (AccountCD, ExecutorCD, FailureCD,
    IncomingDate, ExecutionDate, Executed)
VALUES ('005488', 3, 1, CAST('YESTERDAY' AS DATE),
        CAST('NOW' AS DATE), False);

```

При выполнении данного запроса срабатывает триггер Request_Insert, так как он создан для таблицы Request, активен и имеет реакцию на тип события INSERT. Поскольку вставляемая строка не содержит NULL, то запрос будет выполнен успешно. Если же будет произведена попытка вставки NULL в столбцы внешних ключей таблицы Request или в столбцы IncomingDate и Executed, то сгенерируется исключительная ситуация E_Request, будет выдано сообщение «Insert null value into Request» и вставки не произойдет. При

попытке ввести дату выполнения заявки ранее даты регистрации сгенерируется исключительная ситуация E_I_Request, будет выдано сообщение «ExcecutionDate < IncomingDate» и вставки также не будет.

Для решения задачи в MS SQL Server можно использовать INSTEAD OF-триггер, который реализуем с помощью скрипта

```
/* создание последовательности */
CREATE SEQUENCE Request_Gen;
GO
/* определение триггера */
CREATE TRIGGER Request_Insert ON Request INSTEAD OF INSERT
AS
DECLARE @i INTEGER;
DECLARE CursIns CURSOR
    FOR SELECT AccountCD, ExecutorCD, FailureCD, IncomingDate,
           ExecutionDate, Executed
    FROM INSERTED;
DECLARE @AccountCD VARCHAR(6);
DECLARE @ExecutorCD INTEGER;
DECLARE @FailureCD INTEGER;
DECLARE @IncomingDate DATE;
DECLARE @ExecutionDate DATE;
DECLARE @Executed SMALLINT;
BEGIN
    SET NOCOUNT ON;
    IF EXISTS(SELECT *
              FROM INSERTED
              WHERE (INSERTED.AccountCD IS NULL)
                 OR (INSERTED.ExecutorCD IS NULL)
                 OR (INSERTED.FailureCD IS NULL)
                 OR (INSERTED.IncomingDate IS NULL)
                 OR (INSERTED.Executed IS NULL))
        THROW 51000, 'Error insert null value into Request', 1;
    ELSE
        IF EXISTS(SELECT *
                  FROM INSERTED
                  WHERE INSERTED.ExecutionDate < INSERTED.IncomingDate)
            THROW 51001, 'Error ExecutionDate < IncomingDate', 1;
        ELSE
            BEGIN
                SELECT @i = MAX(RequestCD)+1 FROM Request ;
                EXECUTE('ALTER SEQUENCE Request_Gen RESTART WITH ' + @i);
                OPEN CursIns;
                FETCH NEXT FROM CursIns
                INTO @AccountCD, @ExecutorCD, @FailureCD,
                    @IncomingDate, @ExecutionDate, @Executed;
                WHILE @@FETCH_STATUS = 0
                BEGIN
                    INSERT INTO Request VALUES
                        (NEXT VALUE FOR Request_Gen, @AccountCD,
                         @ExecutorCD, @FailureCD,
```

```

                                @IncomingDate,@ExecutionDate, @Executed);
    FETCH NEXT FROM CursIns
    INTO @AccountCD, @ExecutorCD, @FailureCD,
        @IncomingDate,@ExecutionDate,@Executed;
    END;
    CLOSE CursIns;
    DEALLOCATE CursIns;
END;
END;
GO

```

Приведенный триггер заменяет собой событие вставки строк в таблицу Request. Сначала триггером выполняется проверка существования в таблице INSERTED строк с NULL значениями в столбцах внешних ключей или с датой выполнения заявки, меньшей даты ее регистрации. Если такие строки есть (EXISTS), то вызывается соответствующее исключение с помощью оператора THROW, и вставка строк не выполняется. Если таких строк нет среди добавляемых, то с помощью курсора CursIns обрабатываются строки таблицы INSERTED, и каждая очередная строка добавляется в таблицу Request запросом INSERT с указанием генерируемого значения первичного ключа.

Следует обратить внимание на важную особенность работы триггеров INSTEAD OF. Если триггер INSTEAD OF, определенный для таблицы, выполняет по отношению к таблице какой-либо запрос, который бы снова вызвал срабатывание триггера INSTEAD OF, триггер рекурсивно не вызывается [7]. Вместо этого запрос обрабатывается так, как если бы у таблицы отсутствовал триггер INSTEAD OF, и начинается применение последовательности ограничений и выполнение триггеров AFTER. Так, запрос INSERT в конце кода триггера Request_Insert не вызывает нового срабатывания триггера.

В Oracle реализация рассматриваемого триггера, контролирующего вставку строк в таблицу Request, схожа с реализацией в Firebird:

```

/* создание последовательности */
CREATE SEQUENCE Request_Gen;
/* определение триггера */
CREATE TRIGGER Request_Insert
BEFORE INSERT ON Request FOR EACH ROW
DECLARE
    i INTEGER;
    E_Request EXCEPTION;
    E_I_Request EXCEPTION;
    ErrorMessage VARCHAR2(255);
BEGIN
    IF ((:NEW.AccountCD IS NULL) OR (:NEW.ExecutorCD IS NULL) OR
        (:NEW.FailureCD IS NULL) OR (:NEW.IncomingDate IS NULL) OR
        (:NEW.Executed IS NULL)) THEN

```

```

        RAISE E_Request;
    ELSE
        IF (:NEW.ExecutionDate < :NEW.IncomingDate) THEN
            RAISE E_I_Request;
        ELSE
            :NEW.RequestCD := Request_Gen.NEXTVAL;
        END IF;
    END IF;
EXCEPTION
    WHEN E_Request THEN
        ErrorMessage := 'Error insert null value into Request';
    WHEN E_I_Request THEN
        ErrorMessage := 'Error ExecutionDate < IncomingDate';
END;
/

```

В приведенном триггере предполагается, что генератор Request_Gen создан для таблицы Request изначально и его текущее значение находится в актуальном состоянии. Выполнить переустановку начального значения генератора в соответствии с полученным максимальным значением столбца RequestCD в Oracle, как в предыдущих СУБД, нельзя.

3. Необходимо создать условие ссылочной целостности с помощью триггера, который бы запрещал удаление и изменение значения первичного ключа StreetCD в таблице Street при наличии ссылающихся на него внешних ключей в таблице Abonent. Текст SQL-сценария, реализующего поставленную задачу в Firebird:

```

SET TERM !! ;
CREATE TRIGGER Str_Delete FOR Street
BEFORE DELETE OR UPDATE
AS
BEGIN
    IF ((EXISTS(SELECT FIRST 1 StreetCD
                FROM Abonent
                WHERE StreetCD = OLD.StreetCD)))
    THEN
        BEGIN
            IF (DELETING) THEN
                EXCEPTION Del_Restrict;
            IF (UPDATING) THEN
                EXCEPTION Upd_Restrict;
        END
    END!!
SET TERM ; !!

```

Созданный триггер с именем Str_Delete выполняется перед удалением или обновлением строки в таблице Street. С помощью предиката EXISTS определяется существование строк в таблице Abonent, которые имеют внешние ключи, ссылающиеся на удаляемый или

обновляемый первичный ключ таблицы Street. При определении первой же такой строки анализируется вид события DML. Если это удаление строки, то возникает определенная при создании учебной БД исключительная ситуация Del_Restrict, прерывающая выполнение тела триггера, и удалить из таблицы Street первичный ключ, который имеет ссылающийся на него внешний ключ, становится невозможно. Если это обновление строки, то возникает исключительная ситуация Upd_Restrict, прерывающая выполнение тела триггера, и обновить первичный ключ из таблицы Street, который имеет ссылающийся на него внешний ключ, становится невозможно. Таким образом, созданный триггер реализует правило ссылочной целостности, которое запрещает удаление и обновление первичного ключа родительской таблицы, если имеются ссылающиеся на него внешние ключи дочерней таблицы.

В MS SQL Server реализуем триггер как AFTER-триггер:

```
CREATE TRIGGER Str_Delete ON Street
AFTER DELETE, UPDATE
AS
BEGIN
    IF EXISTS (SELECT *
               FROM Abonent A
               INNER JOIN DELETED D ON A.StreetCD = D.StreetCD)
    BEGIN
        IF EXISTS(SELECT * FROM INSERTED)
        BEGIN
            RAISERROR('Error update', 16, 1);
            ROLLBACK TRANSACTION;
            RETURN;
        END
    ELSE
    BEGIN
        RAISERROR('Error delete', 16, 1);
        ROLLBACK TRANSACTION;
        RETURN;
    END
END
END
GO
```

Приведенный триггер выполняется после удаления либо обновления данных в таблице Street. Если было выполнено обновление данных, то концептуальные таблицы DELETED и INSERTED содержат строки, если удаление — строки присутствуют только в таблице DELETED. С помощью предиката EXISTS проверяется, есть ли в таблице Abonent строки с ключом StreetCD, присутствующим в строках таблицы DELETED. Если такие строки есть, то по наличию строк в таблице INSERTED анализируется вид события DML и с помощью

оператора THROW вызывается соответствующее исключение, а затем транзакция откатывается.

В Oracle для решения задачи создадим триггер следующим запросом:

```
CREATE TRIGGER Str_Delete
BEFORE DELETE OR UPDATE ON Street FOR EACH ROW
DECLARE
    NumRows INTEGER;
    Del_Restrict EXCEPTION;
    Upd_Restrict EXCEPTION;
    ErrorMessage VARCHAR2(255);
BEGIN
    SELECT COUNT(*) INTO NumRows FROM Abonent
    WHERE Abonent.StreetCD = :OLD.StreetCD;
    IF (NumRows > 0) THEN
        IF DELETING THEN
            RAISE Del_Restrict;
        END IF;
        IF UPDATING THEN
            RAISE Upd_Restrict;
        END IF;
    END IF;
EXCEPTION
    WHEN Del_Restrict THEN
        ErrorMessage :=
            'Ограничение удаления записи из родительской таблицы';
    WHEN Upd_Restrict THEN
        ErrorMessage :=
            'Ограничение модификации записи в родительской таблице';
END;
/
```

4. Пусть необходимо создать условие ссылочной целостности с помощью триггера, реализующего добавление строки в справочник улиц Street перед добавлением в таблицу Abonent новой строки с определенным значением внешнего ключа StreetCD, отсутствующим в родительской таблице. Скрипт создания триггера в Firebird:

```
SET TERM !! ;
CREATE TRIGGER Str_Insert FOR Abonent
BEFORE INSERT
AS
DECLARE VARIABLE NumRows INTEGER;
BEGIN
    SELECT COUNT(*) FROM Street S
    WHERE S.StreetCD = NEW.StreetCD INTO :NumRows;
    IF (NumRows = 0) THEN
        INSERT INTO Street (StreetCD) VALUES (NEW.StreetCD);
END!!
SET TERM ; !!
```


Триггер Str_Insert выполняется перед добавлением строки в таблицу Abonent. В первую очередь проверяется наличие строки в родительской таблице Street с первичным ключом StreetCD, равным внешнему ключу новой добавляемой в таблицу Abonent строки. Если соответствующая улица отсутствует в справочнике, выполняется вставка строки в таблицу Street с единственным заполненным полем — полем первичного ключа.

Аналогичный триггер в MS SQL Server реализуем как INSTEAD OF:

```
CREATE TRIGGER Str_Insert ON Abonent INSTEAD OF INSERT
AS
    DECLARE @StreetCD INTEGER;
    DECLARE StreetIns CURSOR
        FOR SELECT StreetCD FROM INSERTED GROUP BY StreetCD;
    BEGIN
        OPEN StreetIns;
        FETCH NEXT FROM StreetIns INTO @StreetCD;
        WHILE @@FETCH_STATUS = 0
        BEGIN
            IF NOT EXISTS (SELECT *
                           FROM Street S
                           WHERE S.StreetCD = @StreetCD)
                INSERT INTO Street (StreetCD) VALUES(@StreetCD);
            FETCH NEXT FROM StreetIns INTO @StreetCD;
        END;
        CLOSE StreetIns;
        DEALLOCATE StreetIns;
        /* вставка строк в целевую таблицу */
        INSERT INTO Abonent SELECT * FROM INSERTED;
    END
GO
```

Приведенный триггер использует курсор для построчного перебора различных кодов улиц StreetCD из виртуальной таблицы INSERTED, которая содержит добавляемые в таблицу Abonent строки. Для каждого кода улицы проверяется наличие родительской записи в таблице Street, и если такая запись не найдена, то выполняется вставка строки в таблицу улиц. Поскольку триггер INSTEAD OF подменяет собой событие добавления, то после выполнения действий по добавлению строк в таблицу улиц Street выполняется добавление данных в таблицу Abonent, для которой создан триггер.

В Oracle реализация триггера для решения задачи практически не отличается от реализации в Firebird:

```
CREATE TRIGGER Str_Insert
BEFORE INSERT ON Abonent FOR EACH ROW
DECLARE
    NumRows INTEGER;
```

```

BEGIN
  SELECT COUNT(*) INTO NumRows FROM Street S
  WHERE S.StreetCD = :NEW.StreetCD;
  IF (NumRows = 0) THEN
    INSERT INTO Street (StreetCD) VALUES (:NEW.StreetCD);
  END IF;
END;
/

```

5. Допустим, необходимо после добавления строк в таблицу начислений или оплат указывать для соответствующего абонента в таблице Abonent, должник он или нет. Следующий скрипт Firebird добавляет в таблицу Abonent столбец Status и создает два триггера:

```

ALTER TABLE Abonent ADD Status VARCHAR(10);
SET TERM !! ;
CREATE TRIGGER Debet_P FOR PaySumma ACTIVE AFTER INSERT
AS
  DECLARE VARIABLE Ostatok Money;
BEGIN
  SELECT
    COALESCE((SELECT SUM(N.NachislSum) FROM NachislSumma N
              WHERE N.AccountCD = NEW.AccountCD),0) -
    COALESCE ((SELECT SUM(P.PaySum) FROM PaySumma P
              WHERE P.AccountCD = NEW.AccountCD),0)
  FROM RDB$DATABASE INTO :Ostatok;
  IF (Ostatok > 0) THEN
    UPDATE Abonent A SET A.Status = 'Должник'
    WHERE A.AccountCD = NEW.AccountCD;
  ELSE
    UPDATE Abonent A SET A.Status = 'Не должник'
    WHERE A.AccountCD = NEW.AccountCD;
END !!
CREATE TRIGGER Debet_N FOR NachislSumma ACTIVE AFTER INSERT
AS
  DECLARE VARIABLE Ostatok Money;
BEGIN
  SELECT
    COALESCE((SELECT SUM(N.NachislSum) FROM NachislSumma N
              WHERE N.AccountCD = NEW.AccountCD),0) -
    COALESCE ((SELECT SUM(P.PaySum) FROM PaySumma P
              WHERE P.AccountCD = NEW.AccountCD),0)
  FROM RDB$DATABASE INTO :Ostatok;
  IF (Ostatok > 0) THEN
    UPDATE Abonent A SET A.Status = 'Должник'
    WHERE A.AccountCD = NEW.AccountCD;
  ELSE
    UPDATE Abonent A SET A.Status = 'Не должник'
    WHERE A.AccountCD = NEW.AccountCD;
END !!
SET TERM ; !!

```

При добавлении новой строки в таблицу оплат PaySumma или в таблицу начислений NachislSumma срабатывает триггер Debet_P или триггер Debet_N, которые заполняют столбец Status значением «Должник», если сумма всех начислений у соответствующего абонента превышает сумму всех его оплат, или значением «Не должник», если сумма всех начислений не превышает сумму всех оплат.

В MS SQL Server скрипт для решения задачи будет выглядеть так:

```
ALTER TABLE Abonent ADD Status VARCHAR(100);
GO
CREATE TRIGGER Debet_P ON PaySumma AFTER INSERT
AS
    DECLARE @Ostatok Money;
BEGIN
    SELECT @Ostatok = COALESCE((SELECT SUM(NachislSum)
    FROM NachislSumma
    WHERE AccountCD = (SELECT ACCOUNTCD FROM INSERTED)),0) -
    COALESCE ((SELECT SUM(PaySum) FROM PaySumma
    WHERE AccountCD = (SELECT ACCOUNTCD
    FROM INSERTED))),0)
    IF (@Ostatok > 0)
        UPDATE Abonent SET status = 'Должник' WHERE AccountCD =
        (SELECT ACCOUNTCD FROM INSERTED);
    ELSE
        UPDATE Abonent SET status = 'Не должник' WHERE AccountCD =
        (SELECT ACCOUNTCD FROM INSERTED);
END
GO
CREATE TRIGGER Debet_N ON NachislSumma AFTER INSERT
AS
    DECLARE @Ostatok Money;
BEGIN
    SELECT @Ostatok = COALESCE((SELECT SUM(NachislSum)
    FROM NachislSumma
    WHERE AccountCD = (SELECT ACCOUNTCD
    FROM INSERTED))),0) -
    COALESCE ((SELECT SUM(PaySum)
    FROM PaySumma
    WHERE AccountCD = (SELECT ACCOUNTCD
    FROM INSERTED))),0)
    IF (@Ostatok > 0)
        UPDATE Abonent SET status = 'Должник'
        WHERE AccountCD = (SELECT ACCOUNTCD FROM INSERTED);
    ELSE
        UPDATE Abonent SET status = 'Не должник'
        WHERE AccountCD = (SELECT ACCOUNTCD FROM INSERTED);
END;
GO
```

Скрипт для Oracle приведен ниже:

```
ALTER TABLE Abonent ADD Status VARCHAR2(10);
/* определение триггеров */
```

```

CREATE TRIGGER Debet_P AFTER INSERT ON PaySumma FOR EACH ROW
DECLARE
    Ostatok NUMERIC(15,2);
BEGIN
    SELECT
        COALESCE((SELECT SUM(N.NachislSum) FROM NachislSumma N
                    WHERE N.AccountCD = :NEW.AccountCD),0) -
        COALESCE ((SELECT SUM(P.PaySum) FROM PaySumma P
                    WHERE P.AccountCD = :NEW.AccountCD),0)
    INTO Ostatok
    FROM DUAL;
    IF (Ostatok > 0) THEN
        UPDATE Abonent A SET A.Status = 'Должник'
        WHERE A.AccountCD = :NEW.AccountCD;
    ELSE
        UPDATE Abonent A SET A.Status = 'Не должник'
        WHERE A.AccountCD = :NEW.AccountCD;
    END IF;
END;
/
CREATE TRIGGER Debet_N AFTER INSERT ON NachislSumma FOR EACH ROW
DECLARE
    Ostatok NUMERIC(15,2);
BEGIN
    SELECT
        COALESCE((SELECT SUM(N.NachislSum) FROM NachislSumma N
                    WHERE N.AccountCD = :NEW.AccountCD),0) -
        COALESCE ((SELECT SUM(P.PaySum) FROM PaySumma P
                    WHERE P.AccountCD = :NEW.AccountCD),0)
    INTO Ostatok
    FROM DUAL;
    IF (Ostatok > 0) THEN
        UPDATE Abonent A SET A.Status = 'Должник'
        WHERE A.AccountCD = :NEW.AccountCD;
    ELSE
        UPDATE Abonent A SET A.Status = 'Не должник'
        WHERE A.AccountCD = :NEW.AccountCD;
    END IF;
END;
/

```

6. Рассмотрим пример простейшего составного (*compound*) триггера Oracle. Пусть необходимо перед добавлением нового лицевого счета в таблицу Abonent проверять, не зарегистрирован ли по указанному адресу уже другой абонент, и, если зарегистрирован, — отменять вставку с выдачей соответствующего сообщения. Если абонента по указанному адресу не зарегистрировано, то новый абонент должен быть добавлен, а количество абонентов на улице должно быть посчитано и записано в таблицу Street. Также количество абонентов на улице должно быть пересчитано при удалении абонента из таблицы Abonent. Решение описанной задачи может

быть выполнено в Oracle с помощью одного составного триггера. Скрипт будет следующим:

```
/* добавление столбца в таблицу Street */
ALTER TABLE Street ADD AbCount INTEGER;
/* определение составного триггера */
CREATE TRIGGER Abonent_Test
FOR INSERT OR DELETE ON Abonent
COMPOUND TRIGGER
    ExistAccount VARCHAR2(6);
    StreetKod INTEGER;
    AbCo1 INTEGER;
    Restrict EXCEPTION;
    ErrorMessage VARCHAR2(255);
BEFORE EACH ROW IS
BEGIN
    IF INSERTING THEN
        SELECT Abonent.AccountCD INTO ExistAccount FROM Abonent
        WHERE Abonent.StreetCD = :NEW.StreetCD
            AND Abonent.HouseNo = :NEW.HouseNo
            AND Abonent.FlatNo = :NEW.FlatNo;
        IF (ExistAccount IS NOT NULL) THEN
            RAISE Restrict;
        END IF;
    END IF;
EXCEPTION
    WHEN Restrict THEN
        ErrorMessage := 'Невозможно добавить строку.' +
            ' По данному адресу уже зарегистрирован абонент';
END BEFORE EACH ROW;
AFTER EACH ROW IS
BEGIN
    IF INSERTING THEN
        StreetKod := :NEW.StreetCD;
    ELSE
        StreetKod := :OLD.StreetCD;
    END IF;
    SELECT COUNT(*) INTO AbCo1
    FROM Abonent A
    WHERE A.StreetCD = StreetKod
    GROUP BY A.StreetCD;
    UPDATE Street S SET S.AbCount = AbCo1
    WHERE S.StreetCD = StreetKod;
END AFTER EACH ROW;
END Abonent_Test;
/
```

Составной триггер Abonent_Test выполняется до и после удаления каждой строки в таблице Abonent. В секции BEFORE EACH ROW перед вставкой строки выполняется проверка наличия зарегистрированного абонента по адресу, указанному для нового добавляемого абонента. Если такой абонент найден, то происходит переход

к блоку исключений секции с выдачей предупреждения и отменой операции. В секции AFTER EACH ROW при добавлении или удалении абонента выполняется подсчет количества абонентов по соответствующей улице, и эта информация заносится в таблицу Street.

Рассмотренные примеры триггеров реализуют различные бизнес-правила и ограничения (см. параграф 1.8). Так первый и четвертый триггеры реализуют активацию операций, второй и третий — ограничения, пятый — вывод, а шестой — ограничение с активацией операций.

6.6.1.3. Модификация и удаление триггера

Триггер представляет собой весьма полезное средство, но в то же время триггеры необходимо очень тщательно отлаживать, так как неправильно написанные триггеры могут привести к серьезным проблемам. При неправильной логике работы триггеров можно легко уничтожить нужные данные и даже целую БД.

Часто возникает необходимость модифицировать ранее созданный триггер DML. Причинами этого могут быть:

- 1) исправление ошибок, допущенных в процессе разработки триггера и нарушающих правильную логику работы системы;
- 2) изменение функциональности уже созданного триггера;
- 3) временное отключение триггера, полезное при разработке и отладке.

Производить модификацию триггера можно в тексте SQL-сценария или путем выполнения отдельного запроса на модификацию, например в SQL-редакторе графической утилиты для работы с БД (IBExpert, dbForge Studio и т. д.).

В Firebird для изменения определения созданного триггера DML необходимо использовать запросы ALTER TRIGGER, CREATE OR ALTER TRIGGER или RECREATE TRIGGER, которые имеют формат

```
{ ALTER TRIGGER имя_триггера
| { CREATE OR ALTER | RECREATE } TRIGGER имя_триггера
  FOR { базовая_таблица | представление } }
[ACTIVE | INACTIVE]
[{ BEFORE | AFTER } <событие1> [OR <событие2> [OR <событие3>] ] ]
[POSITION приоритет_триггера]
[AS <тело_триггера>]
```

Изменения триггера могут быть трех видов:

- 1) только заголовка;
- 2) только тела;
- 3) заголовка и тела.

Если требуется изменить только заголовок триггера, то можно использовать запрос ALTER TRIGGER. Данный запрос требует хотя

бы одного изменяемого атрибута после имени триггера. Любой атрибут заголовка, опущенный в этом запросе, остается неизменным. Если изменяется индикатор фазы, то событие также должно быть указано. Как следует из приведенного выше синтаксиса, запрос ALTER TRIGGER не может связать триггер с другой таблицей, отличной от заданной ранее при создании триггера.

Изменение заголовка триггера может быть использовано, например, для отключения триггера:

```
ALTER TRIGGER Str_Delete INACTIVE;
```

Изменение тела триггера используется, как правило, для преобразования его функциональности. Например, созданный ранее триггер Request_Insert можно изменить, оставив только проверки на NULL:

```
ALTER TRIGGER Request_Insert AS
BEGIN
    IF ((NEW.ExecutorCD IS NULL) OR (NEW.FailureCD IS NULL) OR
        (NEW.IncomingDate IS NULL) OR (NEW.Executed IS NULL))
    THEN
        EXCEPTION E_Request;
    END;
```

Изменение (точнее, замещение) всего триггера (заголовка и тела) следует реализовывать в тексте SQL-сценария. При этом можно использовать запрос ALTER TRIGGER (применение запроса CREATE TRIGGER вызовет ошибку, так как к моменту изменения триггер уже существует).

Часто удобнее использовать запрос CREATE OR ALTER TRIGGER, который создаст триггер, если он еще не существует, или изменит триггер с указанным именем и перекомпилирует его. Имеющиеся зависимости и привилегии сохраняются.

Если требуется заново создать триггер со старым именем, то используется запрос RECREATE TRIGGER. Синтаксис этого запроса такой же, как и запроса CREATE TRIGGER. Если триггер с указанным именем уже существует, то запрос RECREATE TRIGGER пытается удалить его и создать полностью новый объект (что не будет выполнено, если триггер находится в использовании).

В MS SQL Server ранее определенный триггер можно изменить запросом ALTER TRIGGER, который имеет формат

```
CREATE TRIGGER имя_триггера
ON { базовая_таблица | представление }
[ WITH <опция_триггера> [ ,... ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
```

```
[ NOT FOR REPLICATION ]  
AS { <тело триггера> | EXTERNAL NAME <метод> }
```

Как можно заметить, от запроса создания триггера запрос изменения отличается только отсутствием опции WITH APPEND.

Отключить триггер в MS SQL Server путем изменения его заголовка, как, например, в Firebird, нельзя, для этого используется команда DISABLE TRIGGER (формат был приведен в подпараграфе 6.6.1.1).

В Oracle для изменения определения триггеров можно использовать запрос CREATE OR REPLACE, который создает триггер, если он еще не существует, или изменяет триггер с указанным именем и перекомпилирует его.

При создании триггер включается автоматически. Если необходимо временно отключить его, то в Oracle можно применить запрос ALTER TRIGGER формата

```
ALTER TRIGGER имя_триггера { ENABLE | DISABLE};
```

Для удаления триггера любой из рассматриваемых СУБД используется запрос DROP TRIGGER. В Firebird и Oracle запрос имеет формат

```
DROP TRIGGER имя_триггера;
```

Microsoft SQL Server позволяет удалить сразу несколько триггеров одним запросом:

```
DROP TRIGGER имя_триггера [,...];
```

Например, чтобы удалить триггер Str_Delete и правило ссылочной целостности, которое связано с этим триггером, необходимо применить запрос

```
DROP TRIGGER Str_Delete;
```

6.6.2. Триггеры DDL

Триггеры DDL используются для отслеживания выполнения различных операций над объектами БД, таких как создание объекта (CREATE), изменение определения объекта (ALTER), удаление объекта (DROP) [6, 15].

Необходимость в применении триггеров DDL обычно возникает, если нужно:

- 1) предотвратить внесение определенных изменений в схему БД;
- 2) настроить выполнение в БД некоторых действий в ответ на изменения в схеме БД;

3) записывать изменения или события схемы БД в журнал для мониторинга операций пользователей и уведомления администратора о важных событиях, связанных с выполнением команд DDL.

Триггеры DDL существенно расширяют возможности триггеров DML и позволяют реализовать требуемую логику работы приложения.

В Firebird синтаксис создания триггеров DDL во многом похож на синтаксис создания триггеров DML:

```
{ CREATE | RECREATE | CREATE OR ALTER } TRIGGER имя_триггера  
[ ACTIVE | INACTIVE]  
{ BEFORE | AFTER } <событиеDDL>  
[ POSITION приоритет_триггера]  
AS <тело_триггера>
```

где

```
<событиеDDL> ::=  
{ ANY DDL STATEMENT | <событиеDDL1> [{OR <событиеDDL2> }...]
```

в котором

```
<событиеDDL1>, <событиеDDL2> ... ::=  
{ CREATE TABLE | ALTER TABLE | DROP TABLE |  
CREATE PROCEDURE | ALTER PROCEDURE | DROP PROCEDURE |  
CREATE FUNCTION | ALTER FUNCTION | DROP FUNCTION |  
CREATE TRIGGER | ALTER TRIGGER | DROP TRIGGER |  
CREATE EXCEPTION | ALTER EXCEPTION | DROP EXCEPTION |  
CREATE VIEW | ALTER VIEW | DROP VIEW |  
CREATE DOMAIN | ALTER DOMAIN | DROP DOMAIN |  
CREATE ROLE | ALTER ROLE | DROP ROLE |  
CREATE SEQUENCE | ALTER SEQUENCE | DROP SEQUENCE |  
CREATE USER | ALTER USER | DROP USER |  
CREATE INDEX | ALTER INDEX | DROP INDEX |  
CREATE COLLATION | DROP COLLATION | ALTER CHARACTER SET |  
CREATE PACKAGE | ALTER PACKAGE | DROP PACKAGE |  
CREATE PACKAGE BODY | DROP PACKAGE BODY }
```

Примечание. Триггеры DDL создаются, изменяются и удаляются только администратором БД. Тип триггера (BEFORE или AFTER) определяется при создании и не может быть впоследствии изменен.

Триггер DDL может реагировать на любое событие (если объявлено ANY DDL STATEMENT) или на определенные события DDL, такие как удаление таблицы, создание нового индекса, изменение определения хранимой процедуры и т. п.

Триггеры DDL являются частью работы транзакции, в которой выполняется событие изменения схемы БД. Схема будет изменена

только тогда, когда будет успешно подтверждена транзакция. Это следует помнить, в первую очередь, при создании AFTER-триггеров, выполняя в них лишь те действия, которые не предполагают подтвержденного события DDL. Например, нельзя создать таблицу и затем сразу обращаться к ней в триггере AFTER CREATE TABLE.

В случае, если триггер DDL предназначен для предотвращения внесения определенных изменений в схему БД, он при выполнении вызывает пользовательское исключение, и запрос DDL не выполняется. При этом, если выполнение самого запроса DDL приводит к ошибке, то триггер DDL, связанный с таким запросом, не запускается.

Процедуры и триггеры, включенные в PSQL-пакеты (см. параграф 6.4), не могут запускать триггеры DDL для событий {CREATE | ALTER | DROP} {PROCEDURE | FUNCTION}.

В MS SQL Server триггеры DDL срабатывают в ответ на события языка определения данных, а также на некоторые системные ХП, которые выполняют схожие с DDL операции [7]. Запрос создания триггера DDL в MS SQL Server имеет формат

```
CREATE TRIGGER имя_триггера
ON { ALL SERVER | DATABASE }
[ WITH <опция_триггера> [ ,... ] ]
{ FOR | AFTER } { <событиеDDL> | <группа событийDDL> } [ ,... ]
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
AS { <тело триггера> | EXTERNAL NAME <метод> }
```

где

```
<событиеDDL> ::=
{ CREATE_TABLE | ALTER_TABLE | DROP_TABLE |
  CREATE_PROCEDURE | ALTER_PROCEDURE | DROP_PROCEDURE |
  CREATE_FUNCTION | ALTER_FUNCTION | DROP_FUNCTION |
  CREATE_TRIGGER | ALTER_TRIGGER | DROP_TRIGGER |
  CREATE_VIEW | ALTER_VIEW | DROP_VIEW |
  CREATE_TYPE | DROP_TYPE |
  CREATE_INDEX | ALTER_INDEX | DROP_INDEX |
  CREATE_USER | ALTER_USER | DROP_USER ... }
```

В MS SQL Server триггеры DDL реализуются только как AFTER-триггеры. Задаются на двух уровнях: сервер или БД. Список событий, которые могут быть использованы в триггерах DDL, фиксирован на каждом уровне. Каждое событие соответствует запросу DDL или ХП, причем синтаксис команды включает в себя символ подчеркивания _ между ключевыми словами. Помимо отдельных событий, могут быть указаны предопределенные группы, включающие стандартные события, на которые будет реагировать триггер [7]. Например, триггер DDL для группы событий FOR DDL_TABLE_EVENTS

(10018) охватывает события CREATE TABLE, ALTER TABLE и DROP TABLE.

В приведенном выше синтаксисе указаны только основные события изменения схемы БД. С полным списком событий и групп событий, которые могут быть использованы в триггерах DDL, можно ознакомиться на сайте Microsoft [7].

Аналогом концептуальных таблиц INSERTED/DELETED для DDL-триггера является функция EVENTDATA(), выдающая служебную информацию о том, кто, где, когда и что сделал.

Система управления БД Oracle поддерживает создание триггеров DDL запросом вида [2, 18]

```
CREATE [ OR REPLACE ] TRIGGER имя_триггера
{ BEFORE | AFTER }
<событие1> [OR <событие2> [OR <событие3>] ]
ON { [имя_схемы.]SCHEMA | DATABASE }
[ WHEN (<условие>) {
    <тело_блока> | CALL имя_процедуры (<список_параметров>) }]
<тело триггера>
```

где

```
<событие> ::=
{ CREATE | ALTER | RENAME | DROP | TRUNCATE | COMMENT |
  ANALYZE | ASSOCIATE STATISTICS | DISASSOCIATE STATISTICS |
  AUDIT | NOAUDIT | GRANT | REVOKE | DDL }
```

Таким образом, триггеры DDL в Oracle определяются для некоторого события (или событий) по изменению БД без указания конкретного типа объекта (таблица, представление и т. д.). В основном срабатывание триггера распространяется на все типы объектов, но есть ряд исключений:

1) если указано событие ALTER, то запрос ALTER DATABASE не вызывает запуск триггера;

2) если указано событие CREATE, то запросы CREATE DATABASE и CREATE CONTROLFILE не приводят к срабатыванию триггера.

Если событие указано как DDL, то триггер срабатывает при наступлении любого из перечисленных событий, т. е. реагирует на все типы событий DDL.

Триггер может быть создан для схемы БД или для всей БД.

В триггерах DDL доступен для использования ряд системных атрибутов: тип объекта (ORA_DICT_OBJ_TYPE), имя объекта (ORA_DICT_OBJ_NAME), тип операции DDL (ORA_SYSEVENT), CLOB с текстом операции DDL (ORA_SQL_TXT) и т. д. [18, 24].

Рассмотрим несколько примеров триггеров DDL, предназначенных для различных целей.

Например, необходимо создать триггер для обеспечения совместимости имен объектов БД, а именно обеспечить, чтобы имена всех хранимых процедур в БД начинались с префикса 'REP_'. Скрипт на создание такого триггера в Firebird:

```
CREATE EXCEPTION E_Invalid_Rep_Name
  'Неверное имя процедуры (должно начинаться с REP_)';
SET TERM !! ;
CREATE TRIGGER Trig_DDL_Rep BEFORE CREATE PROCEDURE
AS
BEGIN
  IF (RDB$GET_CONTEXT('DDL_TRIGGER', 'OBJECT_NAME')
    NOT STARTING WITH 'REP_') THEN
    EXCEPTION E_Invalid_Rep_Name;
END!!
SET TERM : !!
```

Триггер Trig_DDL_Rep вызывается перед созданием любой ХП. С помощью функции RDB\$GET_CONTEXT он получает имя еще не созданной ХП из переменной OBJECT_NAME, и в случае, если оно начинается не с префикса 'REP_', вызывается предварительно созданное пользовательское исключение E_Invalid_Rep_Name.

В MS SQL Server аналогичный триггер может быть создан так:

```
CREATE TRIGGER Trig_DDL_Rep
ON DATABASE FOR CREATE_PROCEDURE
AS
  DECLARE @EventType VARCHAR(100)
  DECLARE @ObjectName VARCHAR(100)
  DECLARE @ObjectType VARCHAR(100)
BEGIN
  SELECT
    @EventType = EVENTDATA().value(
      '(/EVENT_INSTANCE/EventType)[1]', 'nvarchar(max)'),
    @ObjectName = EVENTDATA().value(
      '(/EVENT_INSTANCE/ObjectName)[1]', 'nvarchar(max)'),
    @ObjectType = EVENTDATA().value(
      '(/EVENT_INSTANCE/ObjectType)[1]', 'nvarchar(max)')
  IF @ObjectName NOT LIKE 'REP_%'
  BEGIN
    RAISERROR (
      'Неверное имя процедуры (должно начинаться с REP_)', 10, 1);
    ROLLBACK;
  END;
END;
GO
```

Данный триггер запускается после создания ХП и с помощью функции EVENTDATA() получает в локальной переменной @ObjectName имя объекта БД, над которым выполняется операция DDL. В при-

мере также показано, как с помощью функции EVENTDATA() можно получить тип объекта и тип события DDL, хотя в данном случае это не требуется. Затем выполняется проверка на то, что имя объекта начинается с 'REP_'. Если это так, то вызывается исключение с помощью оператора RAISEERROR и транзакция откатывается.

В Oracle триггер можно реализовать в следующем виде:

```
CREATE TRIGGER Trig_DDL_Rep BEFORE CREATE ON DATABASE
DECLARE
    E_Invalid_Rep_Name EXCEPTION;
    ErrorMessage VARCHAR2(255);
BEGIN

    IF (ORA_DICT_OBJ_TYPE = 'PROCEDURE')
        AND (ORA_DICT_OBJ_NAME NOT LIKE 'REP_ %') THEN
        RAISE E_Invalid_Rep_Name;
    END IF;
EXCEPTION
    WHEN E_Invalid_Rep_Name THEN
        ErrorMessage :=
            'Неверное имя процедуры (должно начинаться с REP_)';
END;
/
```

Учитывая, что в Oracle триггер DDL создается не для конкретного типа объекта, в коде триггера выполняется проверка не только имени обрабатываемого объекта, но также и его типа. Проверки выполняются с помощью системных атрибутов ORA_DICT_OBJ_TYPE и ORA_DICT_OBJ_NAME, возвращающих имя и тип объекта, над которым совершается операция DDL.

Рассмотрим пример триггера DDL, служащего для протоколирования действий пользователей по изменению схемы БД. Ранее при описании триггеров DML был приведен пример триггера Usr_Action, который фиксировал в таблице Journal имена пользователей, выполнивших изменение данных в таблице NachislSumma (подпараграф 6.6.1). Теперь создадим таблицу-журнал DDL_Log и два триггера Trig_DDL_Log_Before и Trig_DDL_Log_After, которые будут фиксировать в таблице DDL_Log информацию о любых изменениях схемы БД. Скрипт в Firebird будет выглядеть так:

```
CREATE SEQUENCE DDL_Seq;
CREATE TABLE DDL_Log (
    Id BIGINT PRIMARY KEY,
    Usr_Name VARCHAR(20) NOT NULL,
    Time_Date TIMESTAMP NOT NULL,
    DDL_Event VARCHAR(25) NOT NULL,
    Object_Name VARCHAR(31) NOT NULL,
    Sql_Text BLOB NOT NULL,
    Ok BOOLEAN NOT NULL);
SET TERM !! ;
```

```

CREATE TRIGGER Trig_DDL_Log_Before BEFORE ANY DDL STATEMENT
AS
    DECLARE Id TYPE OF COLUMN DDL_Log.ID;
BEGIN
    IN AUTONOMOUS TRANSACTION DO
        BEGIN
            INSERT INTO DDL_Log (ID, Usr_Name, Time_Date,
                                DDL_Event, Object_Name, Sql_Text, Ok)
            VALUES (NEXT VALUE FOR DDL_Seq,
                    CURRENT_USER, CURRENT_TIMESTAMP,
                    RDB$GET_CONTEXT('DDL_TRIGGER','DDL_EVENT'),
                    RDB$GET_CONTEXT ('DDL_TRIGGER', 'OBJECT_NAME'),
                    RDB$GET_CONTEXT ('DDL_TRIGGER', 'SQL_TEXT'),
                    False)
            RETURNING ID INTO Id;
            RDB$SET_CONTEXT ('USER_SESSION', 'Trig_DDL_Log_ID', Id);
        END
    END!!
CREATE TRIGGER Trig_DDL_Log_After AFTER ANY DDL STATEMENT
AS
BEGIN
    IN AUTONOMOUS TRANSACTION DO
        UPDATE DDL_Log SET Ok = True
        WHERE ID = RDB$GET_CONTEXT ('USER_SESSION', 'Trig_DDL_Log_ID');
    END!!
    COMMIT!!
    SET TERM ;!!
    DELETE FROM DDL_Log;
    COMMIT;

```

В данном примере сначала создается таблица DDL_Log для протоколирования изменений схемы БД. Таблица содержит следующие столбцы: Id (уникальное порядковое значение, получаемое с помощью последовательности DDL_Seq), Usr_Name (имя пользователя, внесшего изменения в схему БД), Time_Date (дата и время изменений), DDL_Event (событие DDL), Object_Name (имя объекта БД), Sql_Text (текст SQL-запроса), Ok (признак подтвержденных изменений).

Затем создается триггер Trig_DDL_Log_Before, который в автономной транзакции вносит строку об изменениях в таблицу DDL_Log. Автономная транзакция необходима, чтобы в случае возникновения какой-либо исключительной ситуации в таблице-журнале сохранились изменения. На данном этапе в таблице DDL_Log в столбец Ok сохраняется признак неподтвержденности изменений. После внесения изменений в таблицу ID строки сохраняется в контекстной пользовательской переменной Trig_DDL_Log_ID.

Далее создается триггер Trig_DDL_Log_After, который в автономной транзакции изменяет таблицу DDL_Log, устанавливая признак подтверждения изменений (Ok=True) для соответствующей строки.

Создание триггера Trig_DDL_Log_After вызывает срабатывание первого триггера Trig_DDL_Log_Before, так как он реагирует на лю-

бое событие DDL, и внесение строки в таблицу DDL_Log. Поэтому в конце скрипта удаляются все строки из таблицы DDL_Log, а именно — строка о создании триггера Trig_DDL_Log_After.

Чтобы проверить действие созданных триггеров, от имени пользователя SYSDBA выполним последовательно три запроса:

```
CREATE TABLE Sprav (Number INTEGER, Name VARCHAR(50));
```

```
DROP TABLE Sprav;
```

```
ALTER TABLE Abonent ADD Mail VARCHAR(30);
```

Данные, полученные в результате выполнения запроса

```
SELECT * FROM DDL_Log;
```

и помещенные в таблицу DDL_Log, представлены на рис. 6.18.

Система управления БД MS SQL Server поддерживает DDL-триггеры только фазы AFTER, поэтому решение задачи по протоколированию будем выполнять с помощью одного триггера Trig_DDL_Log. Выполним следующий скрипт:

```
CREATE SEQUENCE DDL_Seq START WITH 1;
GO
CREATE TABLE DDL_Log (
    Id BIGINT PRIMARY KEY,
    Usr_Name NVARCHAR(20) NOT NULL,
    Time_Date DATETIME NOT NULL,
    DDL_Event NVARCHAR(25) NOT NULL,
    Object_Name NVARCHAR(31) NOT NULL,
    Sql_Text NVARCHAR(2000) NOT NULL,
    Ok SMALLINT NOT NULL);
GO
CREATE TRIGGER Trig_DDL_Log
ON DATABASE
FOR DDL_DATABASE_LEVEL_EVENTS
AS
    DECLARE @data XML;
    SET @data = EVENTDATA();
BEGIN
    INSERT DDL_Log (ID, Usr_Name, Time_Date, DDL_Event,
        Object_Name, Sql_Text, Ok)
VALUES (NEXT VALUE FOR DDL_Seq,
    CONVERT(nvarchar(100), CURRENT_USER), GETDATE(),
    @data.value(
        '(/EVENT_INSTANCE/EventType)[1]', 'nvarchar(100)'),
    @data.value(
        '(/EVENT_INSTANCE/ObjectName)[1]', 'nvarchar(max)'),
    @data.value(
        '(/EVENT_INSTANCE/TSQLCommand)[1]', 'nvarchar(2000)'), 1);
END;
GO
```

ID	USR_NAME	TIME_DATE	DDL_EVENT	OBJECT_NAME	SQL_TEXT	OK
1	SYSDBA	08.12.2018 18:25	CREATE TABLE	SPRAV		True
2	SYSDBA	08.12.2018 18:28	DROP TABLE	SPRAV		True
3	SYSDBA	08.12.2018 18:28	ALTER TABLE	ABONENT		True

Рис. 6.18. Данные таблицы DDL_Log

В примере создается таблица DDL_Log для хранения информации о событиях. После этого создается триггер DDL, который реагирует на группу событий DDL_DATABASE_LEVEL_EVENTS, т. е. на любое событие DDL уровня БД. Триггер с помощью контекстных переменных и функции EVENTDATA(), которая в данном примере передается через локальную переменную @data типа XML, заполняет таблицу журнала событий. Признак подтвержденности изменений устанавливается сразу, так как триггер запускается после события DDL, причем в рамках одной с ним транзакции. Для выполнения кода в отдельной транзакции в триггере MS SQL Server можно использовать операторы управления транзакциями BEGIN TRANSACTION, ROLLBACK TRANSACTION и COMMIT TRANSACTION, однако следует относиться к данной возможности с осторожностью из-за высокого риска ошибок.

В СУБД Oracle для создания необходимых объектов можно использовать такой скрипт:

```
CREATE SEQUENCE DDL_Seq START WITH 1;
CREATE TABLE DDL_Log (
    Id NUMBER(19) PRIMARY KEY,
    Usr_Name VARCHAR2(20) NOT NULL,
    Time_Date DATE NOT NULL,
    DDL_Event VARCHAR2(25) NOT NULL,
    Object_Name VARCHAR2(31) NOT NULL,
    Sql_Text VARCHAR2(2000) NOT NULL,
    Ok SMALLINT NOT NULL);
CREATE TRIGGER Trig_DDL_Log_Before BEFORE DDL ON DATABASE
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
    Id DDL_Log.ID %TYPE;
    /* Переменные для получения текста запроса */
    v_ddl CLOB := EMPTY_CLOB;
    sql_text ORA_NAME_LIST_T;
    l_n NUMBER;
BEGIN
    /* Получаем текст запроса DDL */
    l_n := ORA_SQL_TXT(sql_text);
    FOR i IN 1 .. l_n LOOP
        v_ddl := v_ddl || TO_CLOB(TO_CHAR(sql_text(i)));
    END LOOP;
    /* Вставка данных в таблицу аудита */
    INSERT INTO DDL_Log (ID, Usr_Name, Time_Date, DDL_Event,
        Object_Name, Sql_Text, Ok)
    VALUES (DDL_Seq.NEXTVAL,
        SYS_CONTEXT ('USERENV', 'SESSION_USER'),
        CURRENT_TIMESTAMP, ORA_SYSEVENT,
        ORA_DICT_OBJ_NAME, v_ddl, 0)
    RETURNING ID INTO Id;
    DBMS_SESSION.SET_CONTEXT ('CLIENTCONTEXT', 'Trig_DDL_Log_ID', Id);
END;
```

```

/
CREATE TRIGGER Trig_DDL_Log_After AFTER DDL ON DATABASE
BEGIN
    UPDATE DDL_Log SET Ok = 1
    WHERE ID = SYS_CONTEXT ('CLIENTCONTEXT', 'Trig_DDL_Log_ID');
END;
/
DELETE FROM DDL_Log;

```

В данном скрипте Oracle создаются два триггера фаз BEFORE и AFTER, реагирующие на событие DDL, т. е. на любое событие изменения схемы БД. Триггер Trig_DDL_Log_Before заполняет таблицу аудита данными, используя функцию SYS_CONTEXT для получения имени текущего пользователя, контекстную переменную CURRENT_TIMESTAMP для получения даты/времени и системные атрибуты ORA_SYSEVENT, ORA_DICT_OBJ_NAME, ORA_SQL_TXT для получения типа события, имени объекта и текста запроса. После внесения изменений в таблицу ID строки сохраняется с помощью процедуры DBMS_SESSION.SET_CONTEXT в атрибуте Trig_DDL_Log_ID контекста CLIENTCONTEXT. Каждый контекст в Oracle — это именованный набор пар «параметр/значение». Предопределенный контекст USERENV используется для получения атрибутов текущей сессии Oracle, а контекст CLIENTCONTEXT позволяет сеансу создавать собственные атрибуты и задавать им значения [25].

Триггер Trig_DDL_Log_Before работает в рамках автономной транзакции, что задается объявлением для триггера прагмы AUTONOMOUS_TRANSACTION. Эта директива сообщает СУБД, что данный триггер будет выполняться как новая подтранзакция, независимая от родительской транзакции. Работа, выполняемая в триггере, помеченном AUTONOMOUS_TRANSACTION, будет сохранена вне зависимости от того, будет подтверждена или откатена транзакция самого события.

Триггер Trig_DDL_Log_After работает в рамках одной транзакции с операцией DDL и устанавливает в таблице DDL_Log признак подтверждения изменений после успешного завершения операции, вызвавшей его. Строка для изменения определяется путем считывания стандартной функцией SYS_CONTEXT параметра Trig_DDL_Log_ID из контекста CLIENTCONTEXT.

Выполнение задач аудита в рамках автономных транзакций — очень важная задача. Использование для решения такой задачи триггеров, действующих в рамках одной транзакции с событием, не позволяет надежно отследить попытку модифицировать информацию. Триггер должен в любом случае обнаруживать попытку совершить действие, т. е., если пользователь пытается изменить данные, которые он не имеет права менять, триггер должен создавать аудит-запись и прерывать обновление. К сожалению, при

работе в рамках одной транзакции и события и триггера при откате операции, на которую реагирует триггер, откатывается также и аудит-запись, т. е. получается, что ошибка должна реализоваться или о ней не будет никаких сведений. С использованием автономных транзакций появляется возможность надежно зафиксировать в аудит-таблице попытку выполнить операцию, а также откатить саму операцию. Это в конечном счете дает возможность оповестить конечного пользователя, что он не может изменять определенные данные и что есть запись о попытке это сделать.

6.6.3. Триггеры баз данных

Кроме триггеров DML и триггеров DDL, существуют триггеры БД [8, 9]. Они предоставляют возможность производить определяемые пользователем действия при выполнении подключения к БД и отключении от нее, а также при запуске, успешном завершении или откате транзакции.

Синтаксис запроса на создание или изменение триггера БД в Firebird:

```
{CREATE | RECREATE | CREATE OR ALTER} TRIGGER имя_триггера  
[ACTIVE | INACTIVE]  
ON <событие>  
[POSITION приоритет_триггера]  
AS <тело_триггера>
```

где

```
<событие> ::= {CONNECT | DISCONNECT | TRANSACTION START  
               | TRANSACTION COMMIT | TRANSACTION ROLLBACK }
```

События CONNECT и DISCONNECT возникают при выполнении команд подключения к БД и отсоединения от БД, которые были описаны ранее при изучении SQL-скриптов. Событие TRANSACTION START возникает при запуске новой транзакции, т. е. автоматически вместе с первым SQL-запросом или непосредственно после окончания предыдущей транзакции. События TRANSACTION COMMIT и TRANSACTION ROLLBACK возникают при выполнении команд соответственно COMMIT или ROLLBACK, которые означают завершение транзакции (фиксацию или откат) и будут подробно рассмотрены далее.

Если сравнить данный синтаксис с синтаксисом определения триггеров DML, то можно выделить некоторые особенности:

- 1) не указывается конкретная таблица (представление), для которой создается триггер;
- 2) в качестве событий используются подключение к БД, отключение от БД, старт, успешное завершение и откат транзакций;
- 3) отсутствует указание фазы BEFORE или AFTER для событий.

Создавать триггеры БД могут только владелец БД или SYSDBA, в то время как триггер DML может быть определен любым другим пользователем, если он является владельцем таблицы, для которой создается триггер.

Аналогично описанной ранее модификации триггеров DML может быть выполнено изменение активности или изменение тела ранее созданного триггера БД. Однако следует учесть, что событие, с которым ассоциирован триггер БД, не может быть изменено.

В MS SQL Server триггер БД может быть создан для события входа в систему запросом формата

```
CREATE TRIGGER имя_триггера
ON ALL SERVER
[ WITH <опция_триггера> [ ,... ] ]
{ FOR | AFTER } LOGON
AS { <тело триггера> | EXTERNAL NAME <метод> }
```

Триггер на логон срабатывает после фазы аутентификации, но до установления пользовательской сессии, поэтому пытаться выдать в таких триггерах сообщения пользователю бесполезно — он все равно их не увидит.

Триггер на логон следует использовать крайне осторожно, так как в некоторых случаях такой триггер может сделать невозможным вход в систему для всех пользователей, включая членов предопределенной роли сервера sysadmin. Отключить триггер входа в этом случае можно будет только членам роли sysadmin путем подключения с помощью выделенного административного соединения или путем запуска компонента Database Engine в минимальной конфигурации [7].

В СУБД Oracle триггер БД определяется запросом такого же формата, как и триггер DDL, с разницей только в типе событий:

```
<событие> ::=
{ SERVERERROR | LOGON | LOGOFF | STARTUP | SHUTDOWN | SUSPEND |
  DB_ROLE_CHANGE }
```

Таким образом, триггер БД в Oracle может срабатывать на серверные ошибки, подключение/отключение от БД, открытие/закрытие БД, приостановку транзакции, смену приоритета роли.

Следует учесть, что:

1) для событий LOGON, STARTUP, SERVERERROR, SUSPEND и DB_ROLE_CHANGE применимы только AFTER-триггеры;

2) для событий LOGOFF и SHUTDOWN применимы только BEFORE-триггеры;

3) триггеры AFTER STARTUP и BEFORE SHUTDOWN могут использоваться только для уровня DATABASE.

Рассмотрим пример создания триггера БД Test_Connect, заносящего в таблицу Users_Connect имена пользователей, подключающихся к учебной БД, а также дату и время подключения. Скрипт в Firebird будет выглядеть так:

```
CREATE TABLE Users_Connect
  (User_Name VARCHAR(20), Date_Time TIMESTAMP);
SET TERM !! ;
CREATE TRIGGER Test_Connect
ACTIVE
ON CONNECT
AS
BEGIN
  INSERT INTO Users_Connect
  VALUES (CURRENT_USER, CURRENT_TIMESTAMP);
END!!
SET TERM ; !!
```

В результате выполнения данного скрипта при каждом подключении к учебной БД в таблицу Users_Connect будут занесены имя подключающегося пользователя и дата/время подключения.

Для MS SQL Server скрипт будет следующим:

```
CREATE TABLE Users_Connect
  (User_Name VARCHAR(20), Date_Time DATETIME);
GO
CREATE TRIGGER Test_Connect
ON ALL SERVER
FOR LOGON
AS
BEGIN
  INSERT INTO Users_Connect
  VALUES (CURRENT_USER, GETDATE());
END
GO
```

Таблица Users_Connect в MS SQL Server хранит дату и время подключения в столбце с типом DATETIME, а не TIMESTAMP. Тип TIMESTAMP реализован в MS SQL Server, однако отличается от типа данных TIMESTAMP стандарта ANSI. В MS SQL Server тип TIMESTAMP хранит автоматически генерируемое двоичное число, обеспечивающее уникальность в текущей БД, для хранения даты/времени в обычном формате используется тип DATETIME.

В MS SQL Server следует с особой осторожностью относиться к созданию логон-триггера, так как он реагирует не на подключение к конкретной БД, а на подключение к экземпляру сервера БД. В случае ошибки в коде триггера установить соединение с MS SQL Server в обычном режиме будет невозможно.

Скрипт создания в Oracle триггера, реагирующего на подключение к БД:

```
CREATE TABLE Users_Connect
  (User_Name VARCHAR2(20), Date_Time DATE);
CREATE TRIGGER Test_Connect
AFTER LOGON ON DATABASE
BEGIN
  INSERT INTO Users_Connect
  VALUES (SYS_CONTEXT ('USERENV', 'SESSION_USER'),
          CURRENT_TIMESTAMP);
END;
/
```

В заключение приведем скрипт триггера, помещающего в таблицу Executor_Req данные о ФИО каждого исполнителя и количестве невыполненных им заявок на момент подключения к базе данных и дату этого подключения.

```
CREATE TABLE Executor_Req (
  ID_ExeFio INTEGER, Executor_Fio VARCHAR(20),
  Ex_Req_Count INTEGER, DATE_TIME TIMESTAMP);
CREATE TRIGGER Executor_R ACTIVE ON CONNECT
AS
DECLARE Executor_Fio VARCHAR(20) ;
DECLARE Ex_Req_Count INTEGER ;
BEGIN
  DELETE FROM Executor_Req;
  FOR
    SELECT S1.S12, COALESCE(S2.S22, 0)
    FROM (SELECT E.ExecutorCD, E.Fio
          FROM Executor E)
        AS S1(S11, S12)
  LEFT JOIN(SELECT R.ExecutorCD,
                  SUM(IIF(R.ExecutionDate is NULL, 1, 0))
            FROM Request R
            GROUP BY R.ExecutorCD) AS S2(S21, S22)
    ON S1.S11 = S2.S21 INTO :Executor_Fio, :Ex_Req_Count
  DO
  INSERT INTO Executor_Req
  VALUES ( (SELECT COALESCE(MAX(Er.ID_ExeFio) + 1, 1)
            FROM Executor_Req Er),
          : Executor_Fio, :Ex_Req_Count, CURRENT_TIMESTAMP);
END;
```

Контрольные вопросы

1. В чем состоят преимущества использования хранимых процедур в языке SQL?
2. В чем состоят преимущества использования триггеров?

3. Как осуществляется объявление локальной переменной в теле модуля на процедурном SQL?
4. Какие условные операторы используются в процедурном SQL?
5. Что такое курсор? Каковы особенности работы с явным и неявным курсором?
6. Как запомнить значения определенных столбцов изменяемых таблиц в переменных при использовании запросов модификации данных в PSQL?
7. Что такое генератор последовательности? Как его создать, использовать и удалить с помощью средств языка SQL?
8. Что такое исключение? Как оно создается, изменяется и вызывается?
9. С помощью какого оператора осуществляется динамическое выполнение запросов DDL и DML в модуле процедурного SQL? Какие используются формы этого оператора?
10. Что такое SQL-сценарий? Как выполнить создание БД в SQL-сценарии?
11. Что такое триггер DML? Как он создается, модифицируется и удаляется?
12. Как создать триггер БД? Чем триггер БД отличается от триггера DML?
13. В чем состоят отличия семантического характера ХП от триггеров? Как создать, изменить и удалить ХП?
14. Какие виды ХП существуют? Чем они отличаются друг от друга? Как они вызываются?
15. Как выполнить блок кода на процедурном языке без оформления хранимой процедуры или триггера?

Глава 7

ПРОГРАММИРОВАНИЕ ЗАПРОСОВ УПРАВЛЕНИЯ ДОСТУПОМ

В современных многопользовательских информационных системах крайне важной задачей является не только правильно спроектировать структуру БД и манипулировать данными, но также и обеспечить защиту этих данных.

Защита данных — это мероприятия по охране данных от множества возможных угрожающих ситуаций, как преднамеренных, так и случайных.

Среди множества причин возможного разрушения или потери данных могут быть:

- порча или изменение данных неопытным или анонимным пользователем;
- изменение данных авторизованным пользователем, для которого должно быть доступно только чтение данных;
- возникновение конфликта при выполнении двух и более программ;
- неправильное завершение работы приложений;
- изменение БД недопустимым способом обновления и т. д.

Защита БД от подобных проблем реализуется в основном за счет управления доступом к данным БД.

Каждая СУБД должна поддерживать механизм, гарантирующий, что доступ к БД смогут получить только те пользователи, которые имеют соответствующее разрешение. Подсистема санкционированного доступа позволяет предоставить указанным пользователям определенные привилегии на доступ к данным и дать им возможность избирательно и динамически передавать часть выделенных привилегий другим пользователям, отменяя впоследствии эти привилегии, если потребуется [26].

Материал настоящей главы в большей мере ориентирован на администраторов БД. Здесь рассматривается система безопасности, принятая в языке SQL. Излагаются общие правила разграничения доступа пользователей к объектам БД и описываются методы управления доступом в различных СУБД.

7.1. Принципы доступа к данным. Управление пользователями

В любой информационной системе существует конфиденциальная информация, доступ к которой может быть разрешен лишь ограниченному кругу лиц. Безопасность данных является важным аспектом управления БД, поскольку информация, хранимая в БД, представляет собой чрезвычайно ценный ресурс. Поэтому обеспечение безопасности хранимых данных является неотъемлемой частью любой современной СУБД.

Под *безопасностью* данных понимается защита данных от не-санкционированного доступа.

Схема доступа к данным в реляционных СУБД, реализуемая в SQL, базируется на следующих трех принципах:

1) все операции над данными выполняются только от имени конкретного **пользователя**;

2) каждому пользователю разрешается осуществлять операции с одними **объектами** и запрещается делать это с другими;

3) используется система **привилегий**, то есть совокупность прав пользователя на проведение только разрешенных операций над определенными объектами БД.

В общем виде концепция обеспечения безопасности БД заключается в поддержке проверки подлинности (аутентификации) пользователя и его привилегий. Проверка подлинности означает достоверное подтверждение того, что пользователь, пытающийся выполнить некоторое действие, действительно тот, за кого себя выдает.

Проверка привилегий основана на том, что каждому пользователю БД соответствует набор операций, которые он может выполнять по отношению к определенным объектам.

СУБД в своих системных каталогах хранит как описание самих пользователей, так и описание их привилегий по отношению ко всем объектам.

Пользователи СУБД рассматриваются как основные действующие лица, желающие получить доступ к данным или выполнить некоторые действия с ними. Безопасность целесообразно рассматривать в многопользовательской среде, когда существует определенный круг пользователей, имеющих доступ к БД.

В СУБД Firebird логически пользователь — это регистрационная (учетная) запись, доступная во всех БД, обслуживаемых сервером. Зарегистрировать нового пользователя, можно с помощью утилиты командной строки gsec.exe или графического инструмента администрирования БД. Чтобы зарегистрировать нового пользователя, например с помощью IBExpert, необходимо воспользоваться пунктом «Менеджер пользователей» меню «Инструменты». Имя пользователя может иметь длину до 31 символа включительно, пароль — до 31 символа, однако из них для аутентификации используются

только первые 20 символов [6]. Для имен пользователей не важен регистр символов, но пароль является регистрочувствительным.

Создавать новых пользователей в Firebird может только системный администратор SYSDBA. Информация обо всех пользователях, имеющих доступ к серверу Firebird, и их паролях хранится в файле *security3.fdb*. Когда удаленный или локальный клиент соединяется с БД, происходит идентификация пользователя.

Существуют возможность управлять пользователями через SQL-запросы. Для создания учетной записи пользователя применяется запрос

```
CREATE USER user_name PASSWORD 'user_password'
  [FIRSTNAME 'FirstName']
  [MIDDLENAME 'MiddleName']
  [LASTNAME 'LastName'];
```

где *user_name* — имя пользователя; *user_password*, *FirstName*, *MiddleName*, *LastName* — пароль, имя, отчество и фамилия. Ни одно из значений не может содержать букв кириллицы.

В дальнейшем изменить учетную запись пользователя можно с помощью запроса

```
ALTER USER user_name
[SET] {[PASSWORD 'user_password']
      | [FIRSTNAME 'FirstName']
      | [MIDDLENAME 'MiddleName']
      | [LASTNAME 'LastName']};
```

Пользователь с привилегиями SYSDBA может изменять один или несколько атрибутов у существующего пользователя. Непривилегированные пользователи могут изменить только свои атрибуты. С помощью данного выражения не может быть изменено имя пользователя. Если требуется изменить имя пользователя, то старый пользователь должен быть удален (DROP USER), а новый создан:

```
DROP USER user_name;
```

Особенности работы операторов CREATE USER/ALTER USER/DROP USER в Firebird:

- изменения, сделанные запросами CREATE USER/ALTER USER/DROP USER, незамедлительно отражаются в БД учетных записей *security3.fdb*, не дожидаясь завершения транзакции. Откат сделанных изменений не поддерживается;
- управлять учетными записями может пользователь SYSDBA или пользователь, которому была присвоена роль RDB\$ADMIN. Остальные пользователи могут изменять только свою учетную запись через ALTER USER;

- при изменении учетной записи через ALTER USER необходимо указать, по меньшей мере, один изменяемый параметр: PASSWORD, MIDDLENAME, FIRSTNAME или LASTNAME.

Примеры использования выражений CREATE USER/ALTER USER/DROP USER в Firebird:

- создание учетной записи для пользователя mav

```
CREATE USER mav PASSWORD 'test';
```

- изменение имени и фамилии

```
ALTER USER mav FIRSTNAME 'Ivan' LASTNAME 'Ivanov';
```

- изменение пароля

```
ALTER USER mav PASSWORD 'privet';
```

- удаление учетной записи для пользователя mav

```
DROP USER mav;
```

В MS SQL Server система управления пользователями более расширенная, чем в Firebird. Вход в систему БД может быть выполнен с использованием учетной записи пользователя Windows или регистрационного имени входа в MS SQL Server [27]. Для последующего доступа и работы с определенной БД пользователь также должен иметь учетную запись пользователя БД. Для работы с каждой отдельной БД требуется иметь учетную запись пользователя именно для этой БД. Учетную запись пользователя БД можно сопоставить с существующей учетной записью пользователя Windows, группой Windows (в которой пользователь имеет членство), регистрационным именем или ролью.

Управлять безопасностью БД в MS SQL Server можно, например, с помощью среды Management Studio, dbForge Studio for SQL Server или через запросы языка Transact-SQL.

Для управления безопасностью компонента Database Engine применяются запросы языка T-SQL: CREATE LOGIN, ALTER LOGIN и DROP LOGIN. Новое регистрационное имя входа в SQL Server можно создать с помощью запроса [27]

```
CREATE LOGIN login_name
{ WITH option_list1 |
  FROM {WINDOWS [WITH option_list2 [,...]] |
        CERTIFICATE cert_name |
        ASYMMETRIC KEY key_name } };
```

где login_name — создаваемое регистрационное имя.

В секции WITH указываются один или несколько параметров для регистрационного имени либо же указываются в секции FROM сертификат, асимметричный ключ или учетная запись пользователя Windows, связанная с регистрационным именем.

Наиболее важным параметром списка option_list1 является параметр PASSWORD, который задает пароль для данного регистрационного имени (с другими возможными параметрами можно ознакомиться в работе [7]).

Секция FROM может содержать один из следующих параметров:

- WINDOWS — указывает, что данное регистрационное имя соотносится с существующей учетной записью пользователя Windows;
- SERTIFICATE — задает имя сертификата для привязки к данному регистрационному имени;
- ASYMMETRIC KEY — задает имя асимметричного ключа для привязки к данному регистрационному имени (сертификат и асимметричный ключ уже должны присутствовать в БД master).

Изменить свойства учетной записи входа в SQL Server можно с помощью запроса ALTER LOGIN. Запрос позволяет изменить текущий пароль и его конечную дату действия, параметры доступа, БД по умолчанию и язык по умолчанию. Также можно задействовать или отключить определенное регистрационное имя. С синтаксисом запроса и примерами использования можно ознакомиться на официальном сайте Microsoft [7].

Удалить существующее регистрационное имя, которое не ссылается на другие объекты, можно запросом

```
DROP LOGIN login_name;
```

Для управления пользователями БД в Transact-SQL определены запросы CREATE USER, ALTER USER, DROP USER. Для добавления пользователя в текущую БД используется запрос следующего формата:

```
CREATE USER user_name  
[FOR {LOGIN login_name | CERTIFICATE cert_name  
      | ASYMETRIC KEY key_name } ]  
[WITH DEFAULT_SCHEMA = schema_name];
```

где user_name — имя пользователя в БД; login_name — регистрационное имя, для которого создается данный пользователь; cert_name, key_name — соответствующий сертификат и асимметричный ключ. Опция WITH DEFAULT_SCHEMA позволяет указать первую схему, с которой сервер БД будет начинать поиск для разрешения имен объектов для данного пользователя БД. Схемы как объекты БД MS SQL Server подробно описаны в параграфе 7.2.

Примечание. Каждая БД в MS SQL Server имеет своих конкретных пользователей. Поэтому запрос CREATE USER необходимо выполнить

для каждой БД, где должна существовать учетная запись пользователя. Кроме того, для определенной БД регистрационное имя входа в SQL Server может иметь только одного пользователя БД [27].

Для изменения параметров учетной записи пользователя в БД применяется запрос ALTER USER, с синтаксисом которого можно ознакомиться на официальном сайте Microsoft [7]. Запрос позволяет изменить имя пользователя БД, изменить схему пользователя по умолчанию или переопределить пользователя с другим регистрационным именем.

Для удаления пользователя из текущей БД применяется запрос

```
DROP USER user_name;
```

Пользователя, который является владельцем защищаемых объектов (объектов БД), удалить нельзя.

Пример создания регистрационного имени входа и пользователя БД, соответствующего этому имени, в MS SQL Server:

```
CREATE LOGIN IvanovIvan WITH PASSWORD = 'test';
USE ABONENT_MSSQL;
GO
CREATE USER mav FOR LOGIN IvanovIvan;
GO
```

В СУБД Oracle, как и в MS SQL Server, существует понятие схемы в БД. Однако смысл этого понятия здесь несколько другой.

Файлы БД, хранимые и поддерживаемые Oracle, обеспечивают безопасность путем поддержания принадлежащих пользователям таблиц БД и других объектов в отдельной защищенной области [1, 2, 8]. Данная область называется *схемой пользователя*. Все объекты, хранимые в схеме пользователя, называются *объектами схемы*.

На этапе инициализации БД Oracle создаются системные пользователи SYS и SYSTEM, которые имеют права администраторов БД. Привилегированные учетные записи следует использовать крайне осторожно (и редко), не применяя их для создания таблиц или других подобных действий. Например, следует учесть, что для объектов, созданных пользователем SYSTEM, невозможно определить триггеры.

От имени системных пользователей создаются, изменяются и удаляются учетные записи других пользователей БД с ограниченными привилегиями.

Для создания пользователя (схемы) в БД Oracle применяется запрос CREATE USER, упрощенный синтаксис которого следующий:

```
CREATE USER user_name IDENTIFIED BY user_password
[ DEFAULT TABLESPACE <tablespace> ]
[ TEMPORARY TABLESPACE <tablespace> ]
```

```
[ QUOTA { размер | UNLIMITED } ON <tablespace> [, ...] ]  
[ PASSWORD EXPIRE ]  
[ ACCOUNT {LOCK | UNLOCK} ];
```

Запросом обязательно задается имя пользователя и пароль, причем пароль задается без кавычек.

Опция PASSWORD EXPIRE вынуждает пользователя изменить пароль при первом использовании. Опции ACCOUNT LOCK / ACCOUNT UNLOCK позволяют заблокировать или разблокировать учетную запись.

Могут быть указаны табличное пространство по умолчанию, в котором будут находиться создаваемые в данной схеме объекты (DEFAULT TABLESPACE), и табличное пространство, в котором находятся временные сегменты, используемые в процессе сортировки транзакций (TEMPORARY TABLESPACE). С помощью предложения QUOTA могут быть присвоены квоты, указывающие, какой объем памяти пользователя не могут превысить.

Табличное пространство в Oracle — это логическое пространство, которое СУБД использует для хранения объектов БД, таких как индексы, таблицы и т. п. С помощью пространств администратор имеет возможность контролировать использование БД доступного места и оптимизировать быстродействие [28]. При создании БД автоматически создается табличное пространство SYSTEM, которое предназначено для хранения важных внутренних информационных структур (словарь данных, системные ХП и системные сегменты отката). Табличное пространство SYSTEM применяется по умолчанию для всех пользователей БД. Это нежелательно, и рекомендуется создавать дополнительные табличные пространства для размещения пользовательских объектов сразу после создания БД.

Примечание. Табличное пространство SYSTEM используется для хранения важных объектов системной схемы. Не следует создавать пользовательские объекты в нем. Если табличное пространство SYSTEM будет переполнено, то оно подорвет работу СУБД в целом.

Получить список табличных пространств в БД Oracle можно следующим запросом:

```
SELECT TABLESPACE_NAME FROM DBA_TABLESPACES;
```

Подробнее с особенностями создания и использования табличных пространств в Oracle можно ознакомиться в работах [3, 11].

Для изменения учетной записи используется запрос ALTER USER, синтаксис которого во многом повторяет синтаксис запроса создания учетной записи. Упрощенный формат синтаксиса:

```
ALTER USER user_name IDENTIFIED BY user_password  
[ DEFAULT TABLESPACE <tablespace> ]
```

```
[ TEMPORARY TABLESPACE <tablespace> ]
[ QUOTA { размер | UNLIMITED } ON <tablespace> [, ...] ]
[ DEFAULT ROLE {<роль> | ALL EXCEPT <роль> | NONE } ]
[ PASSWORD EXPIRE ]
[ ACCOUNT {LOCK | UNLOCK} ];
```

Данный запрос позволяет изменить пароль указанного пользователя, переназначить табличные пространства, заблокировать/разблокировать пользователя, назначить роль по умолчанию.

Для удаления учетной записи применяется запрос DROP USER формата

```
DROP USER user_name [CASCADE];
```

Опция CASCADE указывает на необходимость удаления всех объектов пользователя перед удалением учетной записи.

Примеры управления пользователями в Oracle:

- создание учетной записи для пользователя mav

```
CREATE USER mav IDENTIFIED BY test;
```

- изменение пароля с необходимостью его смены самим пользователем

```
ALTER USER mav IDENTIFIED BY privet PASSWORD EXPIRE;
```

- удаление учетной записи пользователя mav и всех его объектов

```
DROP USER mav CASCADE;
```

Объекты доступа — это элементы БД, доступом к которым можно управлять (разрешать доступ или защищать от доступа). Обычно объектами доступа являются таблицы, однако ими могут быть и другие объекты БД — представления, ХП и т. д.

Как правило, для обеспечения безопасности требуется, чтобы данные в любой таблице были доступны не всем пользователям, а лишь некоторым из них, а для определенных таблиц необходимо обеспечить выборочный доступ к их столбцам. Некоторым пользователям должен быть запрещен непосредственный (через запросы) доступ к таблицам, но разрешен доступ к этим же таблицам с помощью механизма представлений. Итак, конкретный пользователь обладает конкретными правами доступа к конкретному объекту.

В модели безопасности MS SQL Server для упрощения взаимоотношений между пользователями и объектами дополнительно используются схемы БД (рассматриваются в параграфе 7.2).

Стабильная система управления пользователями — обязательное условие безопасности данных, хранящихся в любой реляционной СУБД.

Привилегии — это системный признак, определяющий операции, которые разрешено выполнять пользователю над конкретными объектами.

Система управления БД от имени конкретного пользователя выполняет операции над БД, т. е. добавляет строки в таблицы (INSERT), удаляет строки (DELETE), обновляет данные в строках таблицы (UPDATE). Она делает это в зависимости от того, обладает ли конкретный пользователь правами на выполнение конкретных операций над конкретным объектом БД. Например, некоторым пользователям может быть разрешено обновление данных в таблицах, в то время как для других допускается лишь выбор данных из этих же таблиц.

Таким образом, в СУБД авторизация доступа осуществляется с помощью привилегий. Установление и контроль привилегий являются обязанностью администратора БД.

7.2. Схемы базы данных в MS SQL Server

Из СУБД, рассматриваемых в настоящем учебнике, схемы как объект БД реализованы только в MS SQL Server. Если физически БД MS SQL Server представлена в одном или более файлах данных, то логически база состоит из произвольного количества схем (schema) [4].

Схема — это коллекция объектов БД, имеющая одного владельца и формирующая одно пространство имен [27]. Имена объектов одного типа в схеме должны быть уникальными. При этом объекты с одинаковыми именами могут присутствовать в различных схемах одной и той же БД.

Схемы используются в модели безопасности компонента Database Engine для упрощения взаимоотношений между пользователями и объектами. Основным назначением схемы является повышение уровня безопасности данных с тем, чтобы предоставить права на использование объектов базы только указанным пользователям или группам пользователей. Таких пользователей называют *принципалами* (principal) [4]. Схема также позволяет выполнить «упаковку» групп объектов, что существенно облегчает манипулирование привилегиями.

Отделение пользователей БД от схем дает значительные преимущества, такие как:

- один принципал может быть владельцем нескольких схем;
- несколько индивидуальных принципалов могут владеть одной схемой посредством членства в ролях или группах Windows;

- удаление пользователя БД не требует переименования объектов, содержащихся в схеме этого пользователя.

Каждая БД в системе имеет следующие схемы по умолчанию, которые создаются автоматически при ее создании [27]:

- guest;
- dbo;
- INFORMATION_SCHEMA;
- sys.

Компонент Database Engine позволяет пользователям, которые не имеют учетной записи пользователя, работать с БД, используя схему guest. Для схемы guest можно применять разрешения таким же образом, как и для любой другой схемы. Кроме того, схему guest можно удалить из любой БД, кроме системных БД master и tempdb.

Каждый объект БД принадлежит одной и только одной схеме, которая является схемой по умолчанию для данного объекта. Схема по умолчанию может быть определена явно или неявно. Если при создании объекта его схема по умолчанию не определена явно, то этот объект принадлежит к схеме dbo. Кроме того, при использовании принадлежащей ему БД регистрационное имя всегда имеет специальное имя пользователя dbo.

Вся информация о схемах содержится в системной схеме INFORMATION_SCHEMA. Схема sys содержит системные объекты, такие как представления каталога.

Схемы можно создавать, изменять и удалять, используя диалоговые средства Management Studio или запросы Transact-SQL [4].

Для создания схемы используется запрос CREATE SCHEMA формата

```
CREATE SCHEMA имя_схемы
[AUTHORIZATION имя_владельца]
[<элемент_схемы> ...];
```

где

```
<элемент_схемы> ::=
{ <определение_таблицы>
| <определение_представления>
| <запрос_GRANT>
| <запрос_REVOKE>
| <запрос_DENY>}
```

Имя схемы должно быть уникальным в текущей БД. Необязательная опция AUTHORIZATION задает владельца схемы.

Сразу в процессе создания схемы в ней можно сформировать объекты (таблицы, представления), а также предоставить (запрос GRANT), удалить (запрос REVOKE) или запретить наследование

полномочий (запрос DENY). Управление полномочиями будет подробно рассмотрено в параграфе 7.3.

Для изменения схемы применяется запрос ALTER SCHEMA формата

```
ALTER SCHEMA имя_схемы  
TRANSFER <элемент_схемы>;
```

Изменение схемы — это всего лишь перемещение объектов из одной схемы в другую. Имя перемещаемого элемента из другой схемы должно состоять из имени схемы и имени объекта, которые разделены точкой.

Удаление пустой схемы (не содержащей объектов) выполняется запросом

```
DROP SCHEMA <имя_схемы>;
```

7.3. Привилегии доступа и передача привилегий

Создание учетной записи пользователя само по себе не дает ему никаких прав на доступ к объектам БД. Безопасность в языке SQL на уровне доступа к таблице (представлению) регулируется с помощью *привилегий доступа*, т. е. списком операций, которые пользователю разрешено выполнять над данной таблицей (представлением). Для установления ограничений на права пользователей при работе с объектами БД используется язык управления доступом к данным, или DCL (Data Control Language).

Язык DCL в различных СУБД может содержать следующие запросы, назначающие, отменяющие и удаляющие полномочия к объектам БД для пользователей и ролей:

- GRANT (предоставление полномочий к защищаемому объекту);
- DENY (отмена полномочия);
- REVOKE (удаление полномочия).

Из рассматриваемых в учебнике СУБД запрос отмены полномочий DENY поддерживается только в MS SQL Server; СУБД Firebird и Oracle поддерживают только запросы GRANT и REVOKE.

Все объекты БД имеют определенный уровень доступа при их создании. Первоначально только создатель объекта БД имеет к нему доступ и только он может передать определенные привилегии доступа другим пользователям, ХП и т. д. При этом обычно системный администратор СУБД имеет доступ ко всем без исключения объектам БД (в Firebird это SYSDBA, в Oracle — SYS и SYSTEM). В связи с этим *настоятельно* рекомендуется не использовать стандартный пароль системного администратора, а сменить его, если

в СУБД он предопределен заранее (например, в Firebird предопределен пароль «masterkey» для SYSDBA).

Примечание. СУБД Firebird не проверяет никакие права доступа для пользователя SYSDBA. В частности поэтому операции над множеством объектов производятся от имени SYSDBA чуть быстрее, чем от имени других пользователей [5].

В СУБД Firebird запрос GRANT может быть использован для операций по передаче привилегий (табл. 7.1):

- передача пользователям, триггерам, ХП или представлениям: SELECT, INSERT, UPDATE, DELETE и REFERENCES привилегий доступа к таблице;
- передача пользователям, триггерам, ХП или представлениям: SELECT, INSERT, UPDATE и DELETE привилегий доступа к представлению;
- передача для роли: SELECT, INSERT, UPDATE, DELETE и REFERENCES привилегий доступа к таблице;
- передача для роли: SELECT, INSERT, UPDATE и DELETE привилегий доступа к представлению;
- передача пользователям права использовать привилегии, переданные конкретной роли, т. е. при подключении к БД обладать привилегиями, определенными соответствующей ролью в БД;
- передача пользователям, триггерам, ХП или представлениям EXECUTE привилегии на выполнение ХП.

Таблица 7.1

Описание привилегий доступа Firebird

Привилегия	Уровень доступа
ALL	Выбор, вставка, обновление, удаление данных и ссылка внешнего ключа другой таблицы на первичный ключ данной таблицы
SELECT	Чтение данных
INSERT	Запись новых данных
UPDATE	Изменение существующих данных
DELETE	Удаление данных
EXECUTE	Выполнение или вызов ХП
REFERENCES	Ссылка внешнего ключа другой таблицы на первичный ключ данной таблицы
Имя_роли	Представляет все привилегии, переданные данной роли

Примечание. Зарезервированное слово ALL обеспечивает механизм для присвоения SELECT, DELETE, INSERT, UPDATE и REFERENCES привилегий одновременно. При этом ALL не передает право на использование роли или привиле-

гию EXECUTE на выполнение ХП.

Запрос GRANT в Firebird имеет синтаксис

```
GRANT {<привилегии> ON [TABLE] {базовая_таблица | представление}  
      TO {<объект> | <список_пользователей> [WITH GRANT OPTION]}  
      | EXECUTE ON PROCEDURE имя_процедуры  
      TO {<объект> | <список_пользователей> [WITH GRANT OPTION]}  
      | <список_ролей>  
      TO {PUBLIC | <список_пользователей> [WITH ADMIN OPTION]} };
```

где

```
<привилегии> ::=  
  {ALL [PRIVILEGES] | <привилегия> | <список_привилегий> | имя_роли}; ,  
<привилегия> ::=  
  {SELECT  
   | DELETE  
   | INSERT  
   | UPDATE [( <список_столбцов> )]  
   | REFERENCES [( <список_столбцов> )]};  
<список_привилегий> ::= [,привилегия [, <список_привилегий> [, ...]]];  
<объект> ::=  
  {PROCEDURE имя_процедуры  
   | TRIGGER имя_триггера  
   | VIEW представление  
   | [ROLE] имя_роли  
   | PUBLIC [, <объект> ...]};  
<список_пользователей> ::=  
  [USER] user_name1 [, [USER] user_name2 ...];  
<список_ролей> ::= имя_роли1 [, имя_роли2 ...]
```

Зарезервированное слово PUBLIC означает, что указанные привилегии доступа к заданному объекту БД становятся *общедоступными* и невозможно отобрать эти привилегии у *конкретного* пользователя. Следует учесть, что PUBLIC используется для обозначения именно всех пользователей БД и переданные с помощью PUBLIC привилегии не будут доступны ХП, триггерам, представлениям и ролям.

Опция WITH GRANT OPTION употребляется для передачи пользователю права передавать привилегии другим пользователям. Опция WITH ADMIN OPTION употребляется для передачи пользователю права передавать полученные роли другим пользователям.

В MS SQL Server запрос GRANT предоставляет принципалам разрешения различных категорий на защищаемые объекты. Это могут быть привилегии на доступ к объектам, а также привилегии на выполнение команд. Привилегии на выполнение команд контролируют возможность создания объектов в БД, самой БД, выполнения процедуры резервного копирования и т. д. Запрос GRANT имеет формат [7, 27]

```
GRANT {ALL [PRIVILEGES] | <привилегия>
      [(<список_столбцов>)] [ ,...n ] }
[ON [<класс>::] <имя_объекта>]
TO <список_пользователей> [WITH GRANT OPTION]
[AS <пользователь>];
```

Опция ALL предоставляет большинство из всех возможных разрешений на защищаемый объект, в версии MS SQL Server 2014 сохранена только для поддержки обратной совместимости.

Конкретное разрешение задается как <привилегия>, могут быть указаны в скобках через запятую конкретные столбцы, на которые предоставляется разрешение. Если список столбцов не указан, то разрешение применимо ко всем столбцам объекта.

В предложении ON указывается имя защищаемого объекта, на который предоставляется разрешение, в параметре <класс> может быть указан класс объекта (например, SCHEMA — для схем БД, OBJECT — для таблиц, представлений и т. д.). Существует три основных класса защищаемых объектов: сервер, БД и схема, которые содержат другие защищаемые объекты, такие как регистрационные имена, пользователи БД, таблицы, ХП и т. д. Предложение ON не используется, если предоставляется системная привилегия на изменение метаданных объекта БД.

Наиболее важные разрешения и защищаемые объекты, к которым они применяются, а также краткое описание предоставляемых каждым разрешением возможностей, приведены в табл. 7.2 [27]. Подробнее с привилегиями MS SQL Server можно ознакомиться на официальном сайте [7].

Таблица 7.2

Разрешения и защищаемые объекты MS SQL Server

Привилегия	Объекты	Описание
SELECT	Таблицы и их столбцы, синонимы, представления и их столбцы, возвращающие табличные значения функции	Предоставляет возможность выборки (чтения) строк. Это разрешение можно ограничить одним или несколькими столбцами
INSERT	Таблицы и их столбцы, синонимы, представления и их столбцы	Предоставляет возможность вставлять столбцы
UPDATE	Таблицы и их столбцы, синонимы, представления и их столбцы	Предоставляет возможность изменять значения столбцов. Это разрешение можно ограничить одним или несколькими столбцами

Привилегия	Объекты	Описание
DELETE	Таблицы и их столбцы, синонимы, представления и их столбцы	Предоставляет возможность удалять столбцы
REFERENCES	Определяемые пользователем функции, таблицы и их столбцы, синонимы, представления и их столбцы	Предоставляет возможность обращаться к столбцам внешнего ключа в родительской таблице, когда пользователь не имеет разрешения SELECT для этой таблицы
EXECUTE	Хранимые процедуры, определяемые пользователем функции, синонимы	Предоставляет возможность выполнять ХП или определенную пользователем функцию
CONTROL	Хранимые процедуры, определяемые пользователем функции, синонимы	Предоставляет возможности, подобные возможностям владельца; получатель имеет практически все разрешения, определенные для защищаемого объекта. Принципал, которому было предоставлено разрешение CONTROL, также имеет возможность предоставлять разрешения на данный защищаемый объект
ALTER	Хранимые процедуры, определяемые пользователем функции, таблицы, представления	Предоставляет возможность изменять свойства (за исключением владения) защищаемых объектов
TAKE OWNERSHIP	Хранимые процедуры, определяемые пользователем функции, таблицы, представления, синонимы	Предоставляет возможность становиться владельцем защищаемого объекта, для которого оно применяется
VIEW DEFINITION	Хранимые процедуры, определяемые пользователем функции, таблицы, представления, синонимы	Предоставляет получателю возможность просматривать метаданные защищаемого объекта
CREATE <объект>	Хранимые процедуры, определяемые пользователем функции, таблицы, представления, синонимы	Предоставляет возможность создавать защищаемые объекты БД. Тип объекта (TABLE, VIEW и т. д.) указывается в привилегии, предложение ON не используется

В СУБД Oracle существуют два типа привилегий: касающиеся системы и объектов [1]. *Привилегии системы* позволяют пользователю выполнять конкретные операции с метаданными объектов БД (создавать таблицы, представления и т. д.) или определенный класс операций. Системные привилегии Oracle назначаются одному или нескольким пользователям или ролям. *Привилегии объектов* разрешают пользователю совершать определенные действия по отношению к данным объектов (например, вставлять данные в таблицу другого пользователя).

Для предоставления привилегий на пользование системой и объектами применяется запрос GRANT следующего формата [1, 2]:

```
GRANT { {ALL PRIVILEGES | <список_системных_привилегий>
        | <список_ролей>}
TO {PUBLIC | <список_ролей>
    | <список_пользователей>} [WITH ADMIN OPTION]
    | {ALL PRIVILEGES
    | <объектная_привилегия> [( <список_столбцов> )]
    [ , ...n ] }
ON { <объект> | DIRECTORY <директория>}
TO {PUBLIC | <список_ролей> | <список_пользователей>}
[WITH HIERARCHY OPTION] [WITH GRANT OPTION] };
```

где

```
<список_пользователей> ::=
    user_name1 [IDENTIFIED BY password1] [, user_name2 [IDENTIFIED
    BY password2]...]
```

Можно присваивать несколько привилегий одним запросом, но привилегии должны относиться к одному типу (объектные, системные или ролевые).

Среди системных привилегий Oracle одной из самых важных является привилегия CREATE SESSION. Если пользователь создан, но данная привилегия ему не предоставлена, то он не сможет подключиться к БД. Также часто предоставляются такие системные привилегии, как CREATE TABLE (создание таблицы в текущей схеме), SELECT ANY TABLE (чтение любой таблицы), EXECUTE ANY PROCEDURE (выполнение любой ХП) и т. д. С полным списком системных привилегий можно ознакомиться в официальной документации компании Oracle [3].

Объектные привилегии могут быть предоставлены как привилегии на конкретный объект БД или как права доступа к объекту-директории, который представляет собой объект Oracle, соответствующий директории в файловой системе. В табл. 7.3 приведены наиболее часто употребляемые объектные привилегии Oracle.

Объектные привилегии Oracle

Привилегия	Объекты	Описание
SELECT	Таблицы, представления (обычные и материализованные), последовательности	Предоставляет возможность чтения строк
INSERT	Таблицы, представления (обычные и материализованные)	Предоставляет возможность вставлять данные
UPDATE	Таблицы, представления (обычные и материализованные)	Предоставляет возможность изменять данные
DELETE	Таблицы, представления (обычные и материализованные)	Предоставляет возможность удалять данные
REFERENCES	Таблицы, представления (не материализованные)	Предоставляет возможность обращаться к столбцам внешнего ключа в родительской таблице
EXECUTE	Хранимые процедуры, функции, пакеты, библиотeki, типы	Предоставляет возможность выполнять процедурные модули
ALTER	Таблицы, последовательности	Предоставляет возможность изменять свойства объектов
DEBUG	Таблицы, представления (не материализованные), хранимые процедуры, функции, пакеты, типы	Предоставляет возможность обращаться к объекту при помощи отладчика
INDEX	Таблицы	Предоставляется право создавать индексы по таблице
UNDER	Представления (не материализованные), типы	Предоставляется право создавать представления-потомки указанного представления
READ	Каталог	Предоставляется привилегия читать файлы из каталога ОС, указанный с помощью полного пути к каталогу
WRITE	Каталог	Предоставляется привилегия записывать файлы в каталог ОС, указанный с помощью полного пути к каталогу

При указании в списке пользователей-получателей привилегий предложения IDENTIFIED BY password происходит установка или изменение пароля, который пользователь должен использовать, чтобы ему была предоставлена привилегия.

Опция WITH HIERARCHY OPTION может употребляться при передаче привилегии SELECT. Позволяет получателю привилегии на объект-родитель получить эти привилегии также на все объекты-потомки.

Существуют определенные ограничения в предоставлении привилегий:

- привилегия или роль не должна предоставляться в запросе GRANT больше одного раза;
- роль нельзя назначить самой себе;
- роли не могут назначаться рекурсивно, т. е. нельзя назначить Роль1 роли Роль2, а потом присвоить Роль2 роли Роль1.

Рассмотрим примеры передачи привилегий в различных СУБД. Пусть на сервере зарегистрированы три пользователя: Ivanov, Petrov и Slonov. Имеются: учебная БД, представление Abonent_View, ХП Factorial и FactorialSet. В СУБД Firebird все объекты созданы предопределенным системным администратором SYSDBA, в MS SQL Server — системным администратором sa, а в Oracle — пользователем MainUser, который наделен полномочиями системного администратора.

Следует помнить, что в Oracle, в отличие от остальных СУБД, в первую очередь следует передать пользователям Ivanov и Petrov полномочия на соединение с БД:

```
GRANT CREATE SESSION TO Ivanov,Petrov;
```

Для передачи пользователю Ivanov *всех привилегий* доступа к таблице Abonent в любой из рассматриваемых СУБД можно выполнить запрос

```
GRANT ALL ON Abonent TO Ivanov;
```

В MS SQL Server запрос будет выполнен, но при этом будет выдано предупреждающее сообщение «Разрешение ALL является устаревшим и поддерживается только для обеспечения совместимости. Оно не означает выдачу ВСЕХ разрешений, определенных для сущности».

После получения привилегий пользователь Ivanov сможет, например, выбрать данные таблицы Abonent с помощью запроса

```
SELECT * FROM Abonent;
```

Такой запрос будет успешно выполнен в Firebird и MS SQL Server, в Oracle попытка выполнить запрос вызовет ошибку «ORA-00942: table or view does not exist».

Следует учитывать, что пользователь Oracle, имеющий соответствующие привилегии, может обращаться к объектам, созданным другим пользователем, только с указанием имени схемы (имени пользователя-создателя) перед именем объекта. Запрос должен выглядеть так:

```
SELECT * FROM MainUser.Abonent;
```

Передача привилегии SELECT на представление Abonent_View пользователю Petrov может быть выполнена таким образом:

```
GRANT SELECT ON Abonent_View TO Petrov;
```

Для передачи в общее пользование права обновлять столбцы Fio и AccountCD таблицы Abonent нужно выполнить запрос

```
GRANT UPDATE (Fio, AccountCD) ON Abonent TO PUBLIC;
```

Для предоставления пользователю нескольких привилегий необходимо перечислить в любом порядке передаваемые привилегии в списке, отделяя их друг от друга запятыми:

```
GRANT SELECT, UPDATE (Fio, AccountCD) ON Abonent TO PUBLIC;
```

Чтобы передать пользователю Ivanov привилегию чтения таблицы Abonent с правом передавать эту привилегию другим пользователям, необходимо употребить запрос

```
GRANT SELECT ON Abonent TO Ivanov WITH GRANT OPTION;
```

Для передачи привилегий *нескольким пользователям* имена пользователей должны быть перечислены через запятую. Например, чтобы передать пользователям Ivanov и Petrov все привилегии на таблицу Abonent, необходим запрос

```
GRANT ALL ON Abonent TO Ivanov, Petrov;
```

При попытке выполнить запрос

```
CREATE TABLE Phone_Sprav  
(AccountCD VARCHAR(6) REFERENCES Abonent(AccountCD),  
  Fio VARCHAR(20),  
  Home_Phone VARCHAR(15),  
  Mobil_Phone VARCHAR(11));
```

от имени Slonov, не имеющего никаких прав на объекты учебной БД, возникнет ошибка. Поэтому необходимо дать пользователю привилегию на создание таблиц:

```
GRANT CREATE TABLE TO Slonov;
```

Однако указанной привилегии будет недостаточно. Чтобы создать таблицу Phone_Sprav, у которой столбец AccountCD является внешним ключом, необходимо наделить пользователя Slonov привилегией REFERENCES:

```
GRANT REFERENCES ON Abonent TO Slonov;
```

Но и после этого попытка создания таблицы пользователем Slonov как в MS SQL Server, так и в Oracle вызовет ошибку.

В MS SQL Server это будет ошибка «Указанное название схемы “dbo” либо не существует, либо отсутствует разрешение для его использования», связанная с тем, что по умолчанию пользователь Slonov создает объекты в схеме dbo и, значит, ему должны быть предоставлены права на изменение данной схемы БД. Такие права можно предоставить запросом

```
GRANT ALTER ON SCHEMA::dbo TO Slonov;
```

После этого запрос создания таблицы пользователем Slonov будет успешно выполнен.

Oracle не требует прав на изменение схемы, так как каждый пользователь по умолчанию создает объекты в своей собственной схеме, однако запрос создания таблицы не будет выполнен из-за ошибки «ORA-01950: no privileges on tablespace SYSTEM». Данная ошибка связана с тем, что по умолчанию пользователь Slonov работает в табличном пространстве SYSTEM, но у него нет квоты на его использование.

Как уже отмечалось, нежелательно создавать пользовательские объекты в табличном пространстве SYSTEM, в реальной БД должны быть созданы дополнительные пространства и назначены пользователям. В рамках работы с учебной БД Oracle по упрощенной схеме допускаем, что дополнительные табличные пространства не создаются и пользователь будет создавать объекты в пространстве SYSTEM.

Дать неограниченную квоту на табличное пространство SYSTEM пользователю Slonov можно путем изменения параметров его учетной записи:

```
ALTER USER Slonov QUOTA UNLIMITED ON SYSTEM;
```

После получения квоты на табличное пространство в Oracle пользователь Slonov сможет успешно выполнить запрос создания таблицы:

```
CREATE TABLE Phone_Sprav  
(AccountCD VARCHAR(6) REFERENCES MainUser.Abonent(AccountCD),
```

```
Fio VARCHAR(20),  
Home_Phone VARCHAR(15),  
Mobil_Phone VARCHAR(11));
```

В СУБД Firebird иногда требуется передача привилегий *хранимым процедурам или триггерам* для доступа к таблицам или представлениям, которые имеют другого владельца. Например, чтобы передать ХП Factorial и триггеру Td_Delete все привилегии доступа на таблицу Abonent, необходимо использовать запрос

```
GRANT ALL ON Abonent TO PROCEDURE Factorial, TRIGGER Td_Delete;
```

Можно задавать привилегии и на выполнение ХП. Для выполнения (вызова) хранимых процедур пользователями или другими объектами БД необходимо передать привилегию EXECUTE на соответствующую ХП.

Рассмотрим пример для Firebird по передаче привилегий пользователю Ivanov на выполнение хранимой процедуры FactorialSet (которая, в свою очередь, вызывает хранимую процедуру Factorial и записывает результат своей работы в таблицу Ftable). Также будем считать, что *никаких других* привилегий пользователь Ivanov в БД не имеет. Рассмотрим последовательность запросов GRANT, которые необходимо выполнить для передачи пользователю Ivanov права на выполнение ХП FactorialSet:

```
GRANT EXECUTE ON PROCEDURE FactorialSet TO Ivanov;
```

Если пользователь Ivanov попытается выполнить хранимую процедуру FactorialSet, то эта попытка потерпит неудачу, так как эта процедура вызывает процедуру Factorial. Таким образом, процедуре FactorialSet требуется привилегия на выполнение ХП Factorial, которую SYSDBA может предоставить, используя запрос

```
GRANT EXECUTE ON PROCEDURE Factorial TO PROCEDURE FactorialSet;
```

Но и в этом случае пользователь Ivanov не сможет выполнить процедуру FactorialSet, так как процедура Factorial использует рекурсивный вызов самой себя, и при этом производится проверка на наличие привилегий доступа при вызове процедуры Factorial. Таким образом, процедуре Factorial требуется привилегия EXECUTE на процедуру Factorial:

```
GRANT EXECUTE ON PROCEDURE Factorial TO PROCEDURE Factorial;
```

Теперь осталась одна проблема, не позволяющая пользователю Ivanov выполнить процедуру FactorialSet. Дело в том, что процедура FactorialSet (созданная SYSDBA) не имеет привилегий на вставку

и удаление строк в таблице Ftable. Следующий запрос GRANT предоставляет необходимые привилегии процедуре FactorialSet:

```
GRANT INSERT, DELETE ON Ftable TO PROCEDURE FactorialSet;
```

Был рассмотрен один вариант передачи привилегий. Можно также перечисленные выше привилегии передать непосредственно пользователю Ivanov, но в Firebird это крайне нежелательно с точки зрения безопасности:

```
GRANT INSERT ON Ftable TO Ivanov;  
GRANT EXECUTE ON PROCEDURE Factorial TO Ivanov;  
GRANT EXECUTE ON PROCEDURE FactorialSet TO Ivanov;
```

В MS SQL Server и Oracle не предусмотрена передача привилегий ХП и триггерам, поэтому все необходимые привилегии должны быть переданы непосредственно пользователю.

Следует отметить, что в графических утилитах для работы с БД различных СУБД для передачи привилегий можно использовать «Менеджер прав» («Менеджер безопасности» и т. п.), который упрощает процедуру наделения различными правами пользователей и других объектов БД, избавляя от непосредственного написания SQL-запросов.

7.4. SQL-роли

Концепцию планирования безопасности упрощенно можно представить следующим образом. Администратор СУБД в соответствии с информационной моделью создает БД. В MS SQL Server дополнительно создаются схемы БД для размещения объектов. В Oracle создаются дополнительные табличные пространства для размещения объектов. Затем с помощью запроса GRANT администратор передает привилегии определенным пользователям в соответствии с их уровнем доступа к данным.

Пусть, например, пользователи 1, 2 и 3 *последовательно* получили привилегию SELECT на таблицу A и привилегию DELETE на таблицу B (рис. 7.1).

Роль представляет собой метод обеспечения безопасности на уровне групп привилегий. Когда нескольким пользователям требуется выполнять подобные действия в определенной БД, то можно воспользоваться ролью, задающей группу пользователей БД, которые могут иметь доступ к одним и тем же объектам БД.

Для объединения привилегий SELECT и DELETE в данном примере используем понятие «роль». На рис. 7.2 представлена иллюстрация предыдущего примера, но при использовании механизма роли.

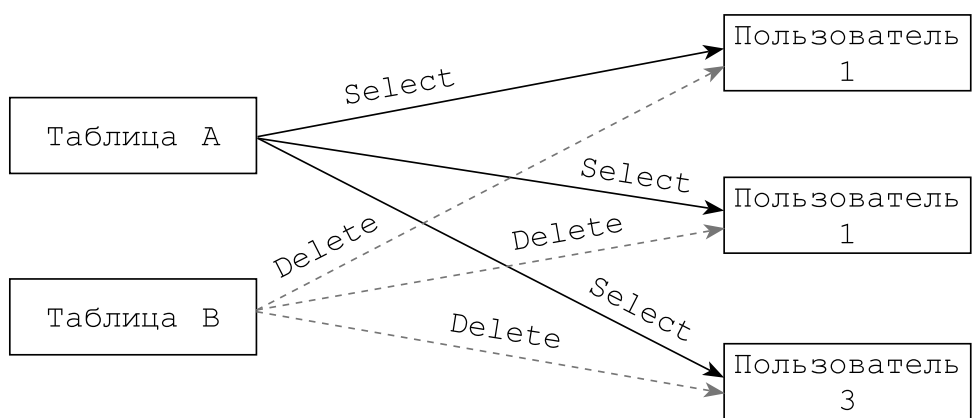


Рис. 7.1. Пример передачи привилегий

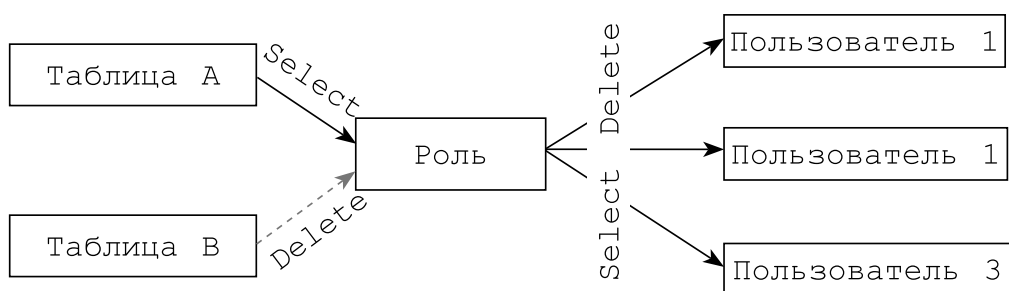


Рис. 7.2. Пример передачи привилегий при использовании роли

Можно сказать, что роль выступает в качестве объекта БД, концентрирующего в себе заданные привилегии доступа к определенным объектам БД.

Архитектура безопасности современных СУБД обычно включает ряд системных ролей, которые имеют специальные явные разрешения, и обеспечивает средства для создания ролей, определяемых пользователями.

В СУБД Firebird роль создается запросом `CREATE ROLE`, а удаляется запросом `DROP ROLE`. Эти запросы имеют синтаксис

```
CREATE ROLE имя_роли;
DROP ROLE имя_роли;
```

Реализация механизма использования роли для передачи привилегий состоит из четырех шагов.

1. Создать роль, используя запрос

```
CREATE ROLE имя_роли;
```

2. Передать этой роли одну или более привилегий, используя запрос

```
GRANT <привилегии> TO имя_роли;
```

3. Передать право на использование этой роли одному или более пользователям с помощью запроса

```
GRANT имя_роли TO <список_пользователей>;
```

При этом роль может быть передана с указанием секции WITH ADMIN OPTION, которое означает, что пользователи, обладающие данной ролью, могут передавать право на использование этой роли другим пользователям.

4. При подключении к БД пользователь, обладающий какой-либо ролью (ролями), должен указать имя этой роли.

Например, чтобы передать пользователю Petrov привилегии SELECT, INSERT и DELETE соответственно на таблицы Abonent, Request и PaySumma, используя роль Petrov_Role, объединяющую в себе перечисленные выше привилегии доступа, необходимо применить запросы:

```
CREATE ROLE Petrov_Role;  
GRANT SELECT ON Abonent TO Petrov_Role;  
GRANT INSERT ON Request TO Petrov_Role;  
GRANT DELETE ON PaySumma TO Petrov_Role;  
GRANT Petrov_Role TO Petrov;
```

Для передачи пользователю Petrov права на передачу роли Petrov_Role другим пользователям необходимо использовать запрос

```
GRANT Petrov_Role TO Petrov WITH ADMIN OPTION;
```

В Firebird существуют контекстные переменные для получения имени пользователя и имени роли, под которыми текущий пользователь подключился к БД. Это переменные CURRENT_USER и CURRENT_ROLE. Переменная CURRENT_ROLE возвращает пустую строку, если текущее соединение не использовало роль.

В Firebird существует системная роль RDB\$ADMIN с привилегиями пользователя SYSDBA. Пользователь SYSDBA может передать свои привилегии другому пользователю.

Следующий пример передает привилегии пользователя SYSDBA пользователям по имени User1 и Admins\ADMINS:

```
GRANT RDB$ADMIN TO User1;  
GRANT RDB$ADMIN TO "Admins\ADMINS";
```

В примере второй пользователь типичен для подлинной аутентификации в Windows.

Примечание. Поскольку Windows проверяет аутентификацию пользователя, БД может быть настроена так, чтобы обеспечить роль RDB\$ADMIN Windows Administrators автоматически.

Система управления БД Oracle поддерживает концепцию ролей, но ее реализация очень отличается от стандарта ANSI SQL. Для управления ролями поддерживаются запросы CREATE ROLE, ALTER ROLE, SET ROLE, DROP ROLE [3].

Запрос создания/изменения роли имеет синтаксис

```
{CREATE | ALTER} ROLE имя_роли  
[ {NOT IDENTIFIED | IDENTIFIED {BY password | EXTERNALLY  
  | GLOBALLY | USING [схема.]имя_пакета} } ];
```

Имя роли в БД не может совпадать с именем пользователя.

Опция NOT IDENTIFIED (установлена по умолчанию) задает, что для прохождения авторизации в БД роли не требуется пароль. Опция IDENTIFIED указывает, что пользователь, которому назначена роль, должен пройти аутентификацию, используя метод, определенный до создания роли с помощью запроса SET ROLE. Аутентификация может быть:

- BY password. Создается локальная роль с аутентификацией по строковому значению пароля;
- EXTERNALLY. Создается внешняя роль, аутентификацию которой производит ОС или сторонний поставщик соответствующих услуг;
- GLOBALLY. Создается глобальная роль, аутентификацию которой производит корпоративная служба каталогов (enterprise directory service);
- USING [схема.]имя_пакета. Создается роль приложения, применяемая только посредством приложения, использующего пакет PL/SQL.

Когда пользователь инициирует соединение, Oracle явным образом назначает этому пользователю роли. Роль (роли), с использованием которой работает сеанс, можно изменить с помощью запроса SET ROLE, если только пользователь имеет соответствующие права доступа. Синтаксис запроса:

```
SET ROLE { <список_ролей> | [ ALL [ EXCEPT имя_роли [, ...] ] | NONE};
```

где

```
<список_ролей> ::= имя_роли1 [ IDENTIFIED BY password1 ]  
[ , имя_роли2 [ IDENTIFIED BY password2 ] ... ]
```

В запросе указываются имена уже существующих ролей, привилегии которых будут присвоены пользователю.

Если у роли есть пароль, то он указывается с помощью предложения IDENTIFIED BY.

Опцией ALL включаются все роли, присвоенные текущему пользователю, в том числе те, которые присвоены через другие роли.

Предложением EXCEPT указывается список ролей, которые исключаются в запросе SET ROLE ALL.

Опцией NONE отключаются все роли, в том числе роль по умолчанию.

Удалить созданную роль можно запросом

```
DROP ROLE имя_роли;
```

Реализация механизма использования роли в Oracle предполагает, что сначала создается роль, а потом ей предоставляются привилегии с помощью запроса GRANT, как обычному пользователю. Если пользователи хотят получить привилегии, предоставляемые ролью, которая защищена паролем, то нужно использовать запрос SET ROLE.

Например, реализация предыдущего примера по передаче пользователю Petrov привилегий на таблицы, используя роль Petrov_Role, в Oracle будет такой:

```
CREATE ROLE Petrov_Role;  
GRANT SELECT ON Abonent TO Petrov_Role;  
GRANT INSERT ON Request TO Petrov_Role;  
GRANT DELETE ON PaySumma TO Petrov_Role;  
GRANT CREATE SESSION TO Petrov_Role;  
GRANT Petrov_Role TO Petrov;
```

Как можно заметить, в Oracle роли передаются права на операции с таблицами, а также право на установление соединения.

Изменим созданную роль, установив пароль, который затем должен будет указать пользователь для использования данной роли:

```
ALTER ROLE Petrov_Role IDENTIFIED BY petr;
```

Доступ к ролям, которые имеют пароли, осуществляется пользователем в текущем сеансе с помощью запроса SET ROLE:

```
SET ROLE Petrov_Role IDENTIFIED BY petr;
```

Проверить привилегии, которыми обладает текущий подключенный пользователь, можно с помощью запроса

```
SELECT * FROM user_sys_privs;
```

В Oracle существует системная роль dba с привилегиями администратора. Передать пользователю права системного администратора можно так:

```
GRANT dba TO MainUser;
```

В СУБД MS SQL Server для управления ролями БД используются запросы CREATE ROLE, ALTER ROLE и DROP ROLE [7, 27].

Для создания новой определяемой пользователем роли БД применяется запрос CREATE ROLE, который имеет следующий синтаксис:

```
CREATE ROLE имя_роли [AUTHORIZATION имя_владельца];
```

Опция AUTHORIZATION задает пользователя БД или роль, которая будет владельцем новой роли. Если опция не задана, то владельцем роли будет пользователь, выполняющий запрос CREATE ROLE.

Для изменения имени роли БД и добавления/исключения членов роли применяется запрос ALTER ROLE формата

```
ALTER ROLE имя_роли  
{ [ ADD MEMBER { <пользователь> | <роль> } ] |  
  [ DROP MEMBER { <пользователь> | <роль> } ] |  
  WITH NAME = новое_имя_роли };
```

Секция ADD MEMBER служит для добавления заданного участника БД к роли БД. Участник может быть пользователем или определяемой пользователем ролью БД. Участник не может быть предопределенной ролью БД или участником на уровне сервера.

Секция DROP MEMBER служит для удаления указанного участника БД из роли БД.

Секция WITH NAME задает новое имя определяемой пользователем роли. Данное имя не должно существовать в БД. Имена предопределенных ролей БД изменить нельзя.

Удалить созданную роль БД можно запросом

```
DROP ROLE имя_роли;
```

Роль, которая является владельцем защищаемых объектов, удалить нельзя. Для удаления сначала нужно изменить владельца этих объектов.

Пример по передаче пользователю Petrov привилегий с помощью роли Petrov_Role в MS SQL Server может быть реализован так:

```
CREATE ROLE Petrov_Role;  
GRANT SELECT ON Abonent TO Petrov_Role;  
GRANT INSERT ON Request TO Petrov_Role;  
GRANT DELETE ON PaySumma TO Petrov_Role;  
ALTER ROLE Petrov_Role ADD MEMBER Petrov;
```

Таким образом, передача роли пользователю осуществляется не с помощью запроса GRANT, как в предыдущих СУБД, а путем изменения определения роли запросом ALTER ROLE.

Изменить имя созданной роли можно следующим запросом:

```
ALTER ROLE Petrov_Role WITH NAME = Abonent_Role;
```

В MS SQL Server полные права на БД содержит в себе предопределенная роль БД db_owner. Пример передачи пользователю:

```
ALTER ROLE db_owner ADD MEMBER User1;
```

7.5. Привилегии на представления

Для контроля доступа к таблицам БД могут быть использованы представления. Представление обычно создается как подмножество столбцов и строк для одной или нескольких таблиц. Из-за этого представление в определенной степени обеспечивает безопасность доступа. Для обеспечения доступа представлений к базовым объектам и последующего доступа пользователей к представлению необходимо использовать запрос GRANT.

При создании представления только для чтения создателю нужны привилегии SELECT к каждой базовой таблице, используемой представлением. Для определения обновляемых представлений создателю нужны привилегии ALL к соответствующим базовым таблицам. В СУБД MS SQL Server и СУБД Oracle также необходима привилегия CREATE VIEW на создание представлений в БД. В MS SQL Server дополнительно необходимо разрешение ALTER в отношении схемы, в которой создается представление.

Пусть, например, системный администратор передал привилегии пользователю Petrov с помощью запроса

```
GRANT ALL ON Abonent TO Petrov;
```

В MS SQL Server и Oracle были переданы привилегии на создание представлений:

```
GRANT CREATE VIEW TO Petrov;
```

Также в MS SQL Server были даны права на изменение схемы:

```
GRANT ALTER ON SCHEMA::dbo TO Petrov;
```

Допустим, что Petrov создает представление Ab_Fio, используя в Firebird и MS SQL Server запрос

```
CREATE VIEW Ab_Fio AS SELECT AccountCD, Fio FROM Abonent;
```

а в Oracle аналогичный запрос

```
CREATE VIEW Ab_Fio AS SELECT AccountCD, Fio FROM MainUser.Abonent;
```

Данный запрос будет выполнен успешно, так как Petrov имеет права на доступ к таблице Abonent. Допустим, Petrov пытается создать еще одно представление и формирует запрос

```
CREATE VIEW Req AS SELECT AccountCD, IncomingDate FROM Request;
```

Данный запрос не будет выполнен, и будет выдано сообщение об ошибке. Например, в Firebird: «No permission for SELECT access to TABLE/VIEW REQUEST» (нет разрешения на SELECT доступ к таблице Request).

Использовать представление только для чтения, т. е. выбирать данные из него, имеют право:

- владелец представления;
- владелец всех базовых таблиц представления;
- администратор БД;
- пользователи, имеющие соответствующие привилегии SELECT для всех базовых таблиц представления;
- пользователи, которым для представления установлены соответствующие привилегии.

На обновляемые представления могут передаваться привилегии INSERT, DELETE и UPDATE. Однако существуют определенные сложности, если представление является обновляемым или включает другие представления или селективные ХП. При обновляемом представлении модификации данных приводят к модификациям данных в базовых таблицах. Поэтому владельцы базовых таблиц должны предоставить пользователю соответствующие права доступа (INSERT, DELETE, UPDATE) к этим таблицам. Если представление обращается к другим представлениям или селективной ХП, то их владельцы должны предоставить пользователю привилегию SELECT на соответствующую ХП или представление. В Firebird в таком случае привилегии к базовым объектам также должны быть предоставлены *самому представлению*.

Привилегии REFERENCES никогда не используется по отношению к представлениям, так как в представлениях не используются внешние ключи.

7.6. Удаление привилегий

Любой пользователь, владеющий объектом, может запретить доступ других пользователей к своим объектам БД. Для удаления привилегий используется запрос REVOKE.

В СУБД Firebird запрос имеет синтаксис

```

REVOKE {[GRANT OPTION FOR] <привилегии>
ON [TABLE] {базовая_таблица | представление}
FROM {<объект> | <список_пользователей> }}
| EXECUTE ON PROCEDURE имя_процедуры
FROM {<объект> | <список_пользователей>}
| [ADMIN OPTION FOR] <список_ролей>
FROM {PUBLIC | <список_пользователей>};

```

где <привилегии>, <объект>, <список_пользователей> и <список_ролей> имеют такой же синтаксис, как и в запросе GRANT.

Зарезервированная секция GRANT OPTION FOR используется для отмены права передавать привилегии другим пользователям. Зарезервированная секция ADMIN OPTION FOR применяется для отмены права передавать роли другим пользователям.

При отмене привилегий запросом REVOKE необходимо учитывать следующие ограничения и правила:

- привилегии могут быть удалены только тем пользователем, который их предоставил;
- привилегии, присвоенные другими пользователями, не затрагиваются;
- удаление конкретной привилегии для пользователя А, которому было дано право передавать ее, автоматически удаляет эту привилегию для всех пользователей, кому она была последовательно передана пользователем А;
- привилегии, переданные для общего пользования (PUBLIC), могут быть отобраны только у PUBLIC, а не у конкретного пользователя или объекта БД.

Начиная с версии Firebird 2.5, добавлена возможность отменить все привилегии в один прием. Для этого используется запрос формата

```

REVOKE ALL ON ALL FROM {<список_пользователей> | <список_ролей>};

```

В СУБД Oracle запрос REVOKE имеет следующий формат [1]:

```

REVOKE { {ALL PRIVILEGES | <список_системных_привилегий> | <список_ролей>}
FROM {PUBLIC | <список_ролей> | <список_пользователей>}
| {ALL PRIVILEGES | <список_объектных_привилегий>}
ON { <объект> | DIRECTORY <директория>}
FROM {PUBLIC | <список_ролей> | <список_пользователей>}
[CASCADE CONSTRAINTS] [FORCE] };

```

Опция CASCADE CONSTRAINTS может использоваться, если удаляются REFERENCES привилегии или привилегии ALL. В этом случае удаляются все ссылочные привилегии, предоставленные явно или неявно через привилегию ALL.

Опция FORCE используется при удалении EXECUTE привилегий на объекты с зависимыми объектами.

В MS SQL Server запрос REVOKE удаляет предоставленное или запрещенное ранее разрешение. Запрос имеет следующий синтаксис:

```
REVOKE [GRANT OPTION FOR]
      { [ALL [PRIVILEGES]] | <список_привилегий> }
      [ON [<класс>::] <объект> ]
      FROM <список_пользователей> [CASCADE] [AS <пользователь>];
```

Опция GRANT OPTION FOR используется для отмены эффекта опции WITH GRANT OPTION в соответствующем запросе GRANT. Это означает, что пользователь или роль все еще будет иметь предоставленные ранее разрешения, но больше не сможет предоставлять их другим пользователям и ролям.

Запрос REVOKE отменяет как «положительные» разрешения, предоставленные запросом GRANT, так и «отрицательные» разрешения, предоставленные запросом DENY (см. далее параграф 7.7). Таким образом, его функцией является нейтрализация указанных ранее разрешений (положительных и отрицательных).

Рассмотрим примеры использования запроса REVOKE.

Пусть необходимо отменить право пользователя Ivanov передавать привилегию на выборку данных из таблицы Abonent другим пользователям. В Oracle опция GRANT OPTION FOR не поддерживается, поэтому рассмотрим пример для СУБД Firebird и MS SQL Server.

В Firebird для этого нужно выполнить запрос

```
REVOKE GRANT OPTION FOR SELECT ON Abonent FROM Ivanov;
```

В MS SQL Server запрос будет таким:

```
REVOKE GRANT OPTION FOR SELECT ON Abonent FROM Ivanov CASCADE;
```

После выполнения этого запроса пользователь Ivanov лишится права передавать другим пользователям привилегию SELECT на таблицу Abonent при сохранении им привилегии на доступ к таблице Abonent.

Особенность отмены права передавать привилегии или роли заключается в каскадности такой отмены. Проиллюстрируем это примером.

Пусть на сервере был зарегистрирован пользователь Slonov, и администратор передал ему привилегии:

```
GRANT ALL ON PaySumma TO Slonov WITH GRANT OPTION;
```

Пользователь Slonov выполнил запрос

```
GRANT SELECT, DELETE ON PaySumma TO Ivanov WITH GRANT OPTION;
```

Пользователь Ivanov, в свою очередь, передал привилегию SELECT пользователю Petrov с помощью запроса

```
GRANT SELECT ON PaySumma TO Petrov;
```

Затем администратор отменил право пользователя Slonov передавать привилегию SELECT другим пользователям (с сохранением права передавать привилегию DELETE). В Firebird отмена выполняется с помощью запроса

```
REVOKE GRANT OPTION FOR SELECT ON PaySumma FROM Slonov;
```

в MS SQL Server запрос выполняется с опцией CASCADE:

```
REVOKE GRANT OPTION FOR SELECT ON PaySumma FROM Slonov CASCADE;
```

В результате каскадного удаления прав:

- пользователь Slonov не имеет права передавать привилегию SELECT;
- пользователь Ivanov вообще лишен привилегии SELECT, а имеет лишь привилегию DELETE (но с опцией WITH GRANT OPTION);
- у пользователя Petrov нет никаких привилегий на доступ к таблице PaySumma. Это произошло, потому что у пользователя Slonov отобрано право передачи привилегии SELECT другим пользователям, а значит, и все действия, которые он произвел, имея это право, также аннулированы (отобрана привилегия SELECT у пользователя Petrov, которую передал ему Ivanov).

Следует отметить, что при передаче пользователям права передавать привилегии может возникнуть проблема множественной передачи. Пусть, например, *каждый* из пользователей — Slonov и Ivanov, имея соответствующие привилегии, выполнили запрос

```
GRANT SELECT ON Request TO Petrov WITH GRANT OPTION;
```

Потом IVANOV выполнил запрос

```
REVOKE SELECT ON Request FROM Petrov;
```

и решил, что после этого пользователь Petrov не имеет доступа к таблице Request. Но у пользователя Petrov сохранилась привилегия, которую передал ему Slonov, и он по-прежнему имеет доступ к таблице Request.

Таким образом, передача привилегий с правом последующей их передачи должна применяться с большой осторожностью или не применяться вовсе.

Чтобы удалить привилегии, переданные какой-либо роли, в любой СУБД можно употребить запрос REVOKE или просто удалить соответствующую роль со всеми ее привилегиями. Например, удаление роли Petrov_ROLE будет выглядеть таким образом:

```
DROP ROLE Petrov_ROLE;
```

7.7. Отмена привилегий

В MS SQL Server, в отличие от других рассматриваемых в учебнике СУБД, может использоваться запрос для отмены привилегий DENY. Запрос DENY запрещает пользователю выполнять указанные действия на указанных объектах. Другими словами, этот запрос удаляет существующие разрешения для учетной записи пользователя, а также предотвращает получение разрешений пользователем посредством его членства в группе или роли, которые он может получить в будущем. Запрос имеет формат [27]

```
DENY {ALL [PRIVILEGES]} | <список_привилегий>  
    [ON [<класс>::] <объект>] TO <список_пользователей>  
    [CASCADE] [AS <пользователь>];
```

Все параметры запроса DENY имеют такое же логическое значение, как и одноименные параметры запроса GRANT.

Дополнительно имеется параметр CASCADE, в котором указывается, что разрешения, запрещенные пользователю, будут также запрещены пользователям, которым он их предоставил. Если параметр CASCADE опущен, и при этом ранее были предоставлены разрешения для соответствующего объекта с использованием предложения WITH GRANT OPTION, то исполнение запроса DENY завершается ошибкой.

Запрос DENY блокирует разрешения, полученные пользователем, группой или ролью посредством их членства в группе или роли. Это означает, что если есть член группы, которому запросом DENY запрещено разрешение, предоставленное для группы, то этот пользователь будет единственным из группы, кто не сможет использовать это разрешение. В то же время если разрешение запрещено для всей группы, то все члены этой группы не смогут пользоваться этим разрешением.

Примечание. В MS SQL Server запрос GRANT можно рассматривать как положительную авторизацию пользователя, а запрос DENY — как отрицательную. Обычно запрос DENY применяется для запрещения разрешений, уже предоставленных для группы (или роли), отдельным членам этой группы.

Отменим, например, для пользователя Ivanov разрешение на обновление столбцов Fio и AccountCD таблицы Abonent, ранее переданное в общее пользование:

```
DENY UPDATE (Fio, AccountCD) ON Abonent TO Ivanov;
```

7.8. Автоматизированное управление доступом

Реализация привилегий SQL требует ручного набора большого объема текста. При наличии работающей схемы привилегий можно генерировать множество скриптов, которые формируют роли и общие полномочия. Возможности современных СУБД позволяют выполнять групповую загрузку полномочий в БД из скрипта безопасности, генерируемого с помощью соответствующей ХП. Скрипт безопасности представляет собой файл, содержащий операторы предоставления/отмены прав и привилегий. Далее приведен пример SQL-сценария для создания такой ХП в Firebird, а также вариант ее использования [5]:

```
SET TERM !! ;
CREATE PROCEDURE Proc1(
  Cmd VARCHAR(6),          /* введите 'G' или 'R' */
  Priv CHAR(10),           /* 'ALL' или 'ANY' или привилегия */
  Usr VARCHAR(31),         /* имя пользователя */
  RoleName VARCHAR(31),    /* роль, существующая или нет */
  GrantOpt SMALLINT,       /* 1 для WITH GRANT (ADMIN) OPTION */
  Create_Role SMALLINT)    /* 1 для создания новой роли */
RETURNS (Perm VARCHAR(80)) /* теоретический оператор полномочия */
AS
  /* для имени таблицы или представления */
  DECLARE VARIABLE RelName VARCHAR (31);
  DECLARE VARIABLE String VARCHAR (80) = '';
  DECLARE VARIABLE Stub VARCHAR (60) = '';
  /* имя пользователя для 'TO' или 'FROM' */
  DECLARE VARIABLE Vusr VARCHAR (31);
  DECLARE VARIABLE Comments CHAR (23) = '/* блочный комментарий */ ';
  BEGIN
    IF (RoleName = '') THEN RoleName = NULL;
    IF (Usr = '') THEN Usr = NULL;
    IF (Priv = '') THEN Priv = NULL;
    /* недостаточно данных для выполнения работы */
    IF ((Priv IS NULL AND RoleName IS NULL) OR Usr IS NULL) THEN EXIT;
    /* если роль введена, работаем с ней */
    IF (RoleName IS NOT NULL) THEN
      BEGIN
        /* создание роли с введенным именем, если Create_Role = 1 */
        IF (Create_Role = 1) THEN
          BEGIN
            Perm = 'CREATE ROLE ' || RoleName || ';';
```

```

        SUSPEND;
        Perm = 'COMMIT;';
        SUSPEND;
        Perm = Comments;
        SUSPEND;
    END
    Vusr = RoleName;
END
/* полномочия только для пользователя */
ELSE Vusr = Usr;
/* выяснение - идет присвоение или отмена привилегий */
IF (Cmd STARTING WITH 'G') THEN
    Stub = 'GRANT ';
ELSE Stub = 'REVOKE ';
IF (RoleName IS NOT NULL) THEN
BEGIN
    IF (Stub = 'GRANT ') THEN
        BEGIN
            String = Stub||RoleName||' TO '||Usr;
            IF (GrantOpt = 1) THEN
                /* предоставление роли пользователю */
                String = String||' WITH ADMIN OPTION;';
            ELSE String = String||';';
        END
    ELSE String = Stub||RoleName||' FROM '||Usr||';';
        Perm = String;
        SUSPEND;
        Perm = Comments;
        SUSPEND;
    END
    /* в данном случае создается отдельно каждая привилегия */
    IF (Priv = 'ANY') THEN
        Stub = Stub||'SELECT, DELETE, INSERT, UPDATE, REFERENCES ON ';
    ELSE
        Stub = Stub||TRIM(Priv)||' ON ';
    /* просмотр всех имен пользовательских таблиц и представлений
    и создание запроса для каждого из них */
    FOR SELECT RDB$RELATION_NAME FROM RDB$RELATIONS
        WHERE RDB$RELATION_NAME NOT STARTING WITH 'RDB$'
            AND RDB$RELATION_NAME NOT STARTING WITH 'MON$'
    INTO :Relname DO
        BEGIN
            String = Stub||TRIM(:Relname)||' ';
            IF (Cmd STARTING WITH 'G') THEN String = String||'TO ';
            ELSE String = String||'FROM ';
            String = String||Vusr;
            IF (Cmd STARTING WITH 'G' AND GrantOpt = 1
                AND RoleName IS NULL) THEN
                String = String||' WITH GRANT OPTION;';
            ELSE String = String||';';
            Perm = String;
            SUSPEND;
        END
END

```

```

Perm = Comments;
SUSPEND;
END!!
SET TERM ; !!

```

Приведенная процедура генерирует скрипт безопасности для предоставления или отмены указанных привилегий нужной роли (если роль не существует, то она создается процедурой) или пользователю (если роль не указана, то привилегии передаются пользователю; если указана — привилегии передаются роли, а роль — пользователю). Если не введено имя пользователя или имя роли либо не указана привилегия, то происходит выход из процедуры.

Переменной Create_Role проверяется необходимость создания роли с данным именем (при условии, что имя введено). Если значение переменной равно 1, то происходит формирование оператора создания роли CREATE ROLE <имя_роли>.

Вспомогательная переменная Vusr принимает значение, равное имени роли (Rolename) или имени пользователя (Usr), если имя роли не введено. Если значение переменной Cmd равно G, то формируется запрос на предоставление привилегии (Stub=GRANT), иначе — на отмену привилегии (Stub=REVOKE).

Переменная GrantOpt отвечает за право передавать привилегии или роль. При указанном имени роли формируется запрос вида GRANT <имя_роли> TO user_name WITH ADMIN OPTION (если GrantOpt=1), либо GRANT <имя_роли> TO user_name (если GrantOpt=0), либо REVOKE <имя_роли> FROM user_name.

Затем происходит определение привилегий: либо ALL, либо конкретная привилегия, либо SELECT, DELETE, INSERT, UPDATE, REFERENCES. Выполняется просмотр имен пользовательских таблиц и представлений из системной таблицы RDB\$RELATIONS (см. приложение Б), и для каждого из них формируется один из запросов:

- 1) GRANT <привилегия> TO <роль> WITH GRANT OPTION (если GrantOpt = 1, имя роли указано);
- 2) GRANT <привилегия> TO <роль> (если GrantOpt = 0, имя роли указано);
- 3) GRANT <привилегия> TO user_name WITH GRANT OPTION (если GrantOpt = 1, имя роли не указано);
- 4) GRANT <привилегия> TO user_name (если GrantOpt = 0, имя роли не указано)

Следующий скрипт Firebird создает скрипт безопасности путем вызова ХП Proc1 и выполняет его:

```

OUTPUT d:\Safety_Script.sql;
SELECT * FROM Proc1('G','ALL','User1','Main',0,1);
COMMIT;

```

```
OUTPUT;  
INPUT d:\Safety_script.sql;  
COMMIT;
```

Первая строка перенаправляет НД запроса SELECT в указанный файл скрипта безопасности d:\Safety_Script.sql. Вторая строка автоматически создает роль Main, предоставляет эту роль пользователю User1, а затем устанавливает роли требуемые полномочия. Команда OUTPUT без параметров закрывает файл, созданный предыдущей командой OUTPUT. Команда INPUT выполняет созданный скрипт безопасности. Скрипт безопасности, сформированный для учебной БД, в данном случае будет иметь вид

```
CREATE ROLE Main;  
COMMIT;  
/* блочный комментарий */  
GRANT Main TO User1;  
/* блочный комментарий */  
GRANT ALL ON STREET TO Main;  
GRANT ALL ON NACHISLSUMMA TO Main;  
GRANT ALL ON SERVICES TO Main;  
GRANT ALL ON DISREPAIR TO Main;  
GRANT ALL ON EXECUTOR TO Main;  
GRANT ALL ON ABONENT TO Main;  
GRANT ALL ON PAYSUMMA TO Main;  
GRANT ALL ON REQUEST TO Main;  
/* блочный комментарий */
```

Возможен также вариант ХП, непосредственно из которой с помощью оператора EXECUTE STATEMENT может выполняться установка/отмена необходимых полномочий [5].

Контрольные вопросы

1. Что может быть причиной разрушения или потери данных в БД?
2. Какие общие требования предъявляются к безопасности реляционных СУБД? На каких принципах базируется схема доступа к данным?
3. Какие привилегии доступа существуют в БД?
4. Как выполнить передачу привилегий с помощью SQL-запроса?
5. Из каких шагов состоит реализация механизма использования SQL-роли?
6. Как пользователь SYSDBA может передать свои права другому пользователю?
7. Каким образом осуществляется отмена привилегий на языке SQL?
8. Какие ограничения и правила необходимо учитывать при отмене привилегий оператором REVOKE?
9. Какие особенности связаны с передачей привилегий при использовании представлений?
10. Как можно реализовать автоматизированное управление доступом в Firebird?

Глава 8

УПРАВЛЕНИЕ ТРАНЗАКЦИЯМИ

Большинство современных СУБД являются многопользовательскими, в связи с чем возникает потребность в организации совместной работы нескольких пользователей с одной и той же совокупностью данных.

Система управления данными, которые изменяются множеством одновременно работающих пользователей, является уязвимой со стороны большого количества проблем целостности данных, если будет работать без какого-либо контроля.

Для обеспечения параллельного доступа и целостности данных многопользовательские СУБД используют модель управления, которая изолирует каждую задачу внутри уникального контекста. Любые действия, выполняемые с БД, осуществляются в контексте *транзакций*.

В предыдущих главах уже косвенно упоминались транзакции, данная глава полностью посвящена описанию механизма транзакций в СУБД Firebird, Oracle, MS SQL Server.

Область организации транзакций и управления ими очень широка. Поддержание механизма транзакций является показателем уровня развитости СУБД. Транзакции, наряду с механизмами разделения доступа к данным, описанным в гл. 7, обеспечивают защиту данных БД. Они помогают предотвратить потери данных, например, в таких ситуациях:

- завершение работы программ при системном сбое, когда БД остается в непредсказуемом состоянии;
- возникновение конфликта при выполнении двух и более программ, конкурирующих за одни и те же данные, и т. д.

В различных СУБД по-разному организован механизм транзакций. Мы ограничимся рассмотрением ключевых концепций и наиболее общих вопросов управления транзакциями преимущественно на примере Firebird в пределах языка SQL, затрагивая при этом и особенности реализации в Oracle и MS SQL Server.

В данной главе приводится определение транзакции и ее свойств, рассматривается механизм сохранения и отката транзакций, описывается использование механизма транзакций для восстановления системы при повреждениях. Вводятся понятие параллельности в ра-

боте БД и методы управления параллельностью с использованием механизмов блокировок и версионности.

Материал настоящей главы, как и предыдущей, в большей мере ориентирован на администраторов БД.

8.1. Понятие транзакции

Транзакция — это группа операций обработки данных, выполняемых как некоторое неделимое действие над БД, осмысленное с точки зрения пользователя. Запись данных в БД производится только при успешном выполнении всех операций группы. Если хотя бы одна из операций группы завершается неуспешно, то БД возвращается к тому состоянию, в котором она была до выполнения первой операции группы. Транзакция реализует некоторую прикладную функцию, например перевод денег с одного счета на другой при оплате услуг по безналичному расчету (снять и положить). Операция перевода денег с банковского счета абонента, производящего оплату безналичным путем, на счет организации, предоставившей услуги, должна составлять единую транзакцию. Иначе может возникнуть ситуация, когда первый SQL-оператор переведет деньги на другой счет, а второй, выполняющий снятие их со счета, не доведет дело до конца из-за непредвиденного сбоя.

Транзакции характеризуются четырьмя классическими свойствами: атомарности, согласованности, изолированности, долговечности (прочности) — Atomicity, Consistency, Isolation, Durability (ACID) [29]. Поэтому часто транзакции называют ACID-транзакциями. Опишем эти свойства подробнее.

Свойство атомарности выражается в том, что транзакция должна быть выполнена в целом или не выполнена вовсе. СУБД гарантирует невозможность фиксации некоторой части действий из транзакции в БД.

Свойство согласованности гарантирует, что по мере выполнения транзакций данные переходят из одного согласованного состояния в другое, т. е. транзакция не разрушает взаимной согласованности данных.

Свойство изолированности означает, что конкурирующие за доступ к БД транзакции физически обрабатываются последовательно, изолированно друг от друга, но для пользователей это выглядит так, как будто они выполняются параллельно. Например, для любых двух транзакций T1 и T2 справедливо следующее утверждение: T1 сможет увидеть обновление T2 только после выполнения T2, а T2 сможет увидеть обновление T1 только после выполнения T1.

Свойство долговечности означает, что если транзакция завершена успешно, то изменения в данных, произведенные в ней, не могут

быть потеряны ни при каких обстоятельствах (даже в случае последующих ошибок или сбоя системы).

Использование транзакций дает ряд преимуществ:

1) механизм транзакций позволяет обеспечить логическую целостность данных в БД. Другими словами, транзакции — это логические единицы работы, после выполнения которых БД остается в целостном состоянии;

2) транзакции являются единицами восстановления данных. Восстанавливаясь после сбоев, система ликвидирует следы транзакций, не успевших успешно завершиться в результате программного или аппаратного сбоя;

3) механизм транзакций обеспечивает правильность работы в многопользовательских системах при параллельном обращении нескольких пользователей к одним и тем же данным.

Рассмотрим подробнее особенности языка SQL для управления транзакциями.

8.2. Восстановление данных

Восстановление в СУБД означает способность возвращения БД в правильное состояние, если какой-то сбой сделал текущее состояние неправильным или подозрительным. Основной принцип восстановления — избыточность. Избыточность обеспечивается фиксацией транзакций, а восстановление — откатом транзакций.

Язык СУБД, отвечающий за управление транзакциями, называется языком обработки транзакций (Transaction Processing Language — TPL).

В Firebird системный компонент СУБД, обеспечивающий ACID-свойства транзакций, называется *администратором транзакций*. Он содержит команды COMMIT и ROLLBACK.

Команда COMMIT завершает текущую транзакцию, выполняя фиксацию сделанных изменений в БД. Иногда говорят, что команда COMMIT фиксирует транзакцию.

Фиксация транзакции — это действие, обеспечивающее запись на диск изменений в БД, сделанных при выполнении транзакции. До тех пор, пока транзакция не зафиксирована, существует возможность аннулирования этих изменений, восстановления БД в то состояние, в котором она была на момент начала транзакции. Фиксация текущей транзакции означает:

- что все результаты выполнения транзакции становятся постоянными;
- результаты будут видны другим транзакциям (до этого момента все данные, затрагиваемые транзакцией, будут «видны» пользователю в состоянии на начало текущей транзакции).

Команда COMMIT имеет синтаксис

COMMIT [WORK] [RETAIN];

Слово WORK является необязательным, поэтому команда COMMIT WORK полностью аналогична команде COMMIT. Команда COMMIT (иногда называемая *жестким подтверждением*) освобождает все физические ресурсы, связанные с транзакцией. Использование ключевого слова RETAIN после команды COMMIT означает, что транзакция зафиксируется, но физические ресурсы не будут освобождены. Такое подтверждение иногда называют *мягким подтверждением*. Оно может быть полезным при выполнении логической задачи, которая включает в себя множество повторений похожих операций. Например, мягкое подтверждение сохраняет открытые в настоящий момент курсоры для выбранных наборов.

Команда ROLLBACK выполняет откат транзакции. *Откат транзакции* — это действие, обеспечивающее аннулирование всех изменений данных, которые были сделаны операторами SQL в теле текущей незавершенной транзакции. Каждый оператор в транзакции выполняет свою часть работы, но для успешного завершения всей работы в целом требуется безусловное завершение всех ее операторов. Команда ROLLBACK имеет синтаксис

ROLLBACK [WORK] [RETAIN] [TO [SAVEPOINT] имя_точки_сохранения];

Как и COMMIT, команда ROLLBACK освобождает ресурсы на сервере. При использовании ROLLBACK RETAIN выделенные ресурсы не освобождаются, курсоры сохраняются. Такой откат следует использовать с большой осторожностью, поскольку отдельные вызовы откатов производятся в ответ на исключение некоторого вида. В таком случае использование ROLLBACK RETAIN также сохранит и причины исключения.

Новая транзакция начинается с начала каждого сеанса работы с БД. Далее все выполняемые SQL-операторы будут входить в одну транзакцию до тех пор, пока не будет выполнена команда COMMIT или ROLLBACK, т. е. чтобы начать транзакцию, не требуется выполнять никаких специальных действий. Транзакция начинается автоматически вместе с первым SQL-оператором или непосредственно после окончания предыдущей транзакции.

Логически транзакция должна объединять только выполнение взаимосвязанных операций. Так, если делать транзакции «очень большими», состоящими из длинной последовательности не связанных между собой операторов, то любой сбой, автоматически выполняющий откат транзакции, повлияет на отмену действий, которые могли бы быть успешно завершены при более «коротких» транзакциях.

Рассмотрим транзакцию «Изменить оплату с кодом 6 с 20 на 100, а затем изменить дату оплаты с '06/13/2016' на '06/11/2016'». Предполагается, что пользователь монопольно вводит запросы SQL последовательно один за другим, пользуясь некоторым интерактивным средством. Например, рассмотрим на примере работы в Firebird с использованием IBExpert. Тело транзакции будут составлять запросы:

```
UPDATE PaySumma SET PaySum = 100
WHERE PayFactCD = 6;
UPDATE PaySumma SET PayDate = '06/11/2016'
WHERE PayFactCD = 6;
...К этому моменту никаких ошибок не обнаружено...
COMMIT WORK;
```

Последняя команда фиксирует изменения в БД, достигнутые в результате выполнения запросов SQL, составляющих транзакцию.

Вновь рассмотрим ту же самую транзакцию, но предположим, что в процессе ввода запросов SQL пользователь допустил ошибку, указав в последнем запросе ошибочную дату, например:

```
UPDATE PaySumma SET PaySum = 100
WHERE PayFactCD = 6;
UPDATE PaySumma SET PayDate = '05/11/2016'
WHERE PayFactCD = 6;
```

Поскольку произошла ошибка, используется команда ROLLBACK WORK. Все изменения в БД, декларированные запросами внутри транзакции, отменяются, и БД возвращается к состоянию на момент начала транзакции.

Возможны два варианта завершения транзакции:

1) транзакция фиксируется, если все операторы выполнены успешно и в процессе выполнения транзакции не произошло никаких сбоев программного или аппаратного обеспечения;

2) БД должна быть возвращена в исходное состояние, если в процессе выполнения транзакции случилось то, что делает невозможным ее нормальное завершение.

В общем случае транзакция завершается одним из четырех возможных путей:

1) использование команды COMMIT в случае успешного завершения транзакции;

2) применение команды ROLLBACK в случае прерывания транзакции;

3) успешное завершение программы, в которой была инициирована текущая транзакция, означает успешное завершение транзакции (как будто была использована команда COMMIT);

4) ошибочное завершение программы прерывает транзакцию (как будто была использована команда ROLLBACK).

На рис. 8.1 в графическом виде представлены типичные транзакции, иллюстрирующие четыре перечисленные ситуации [17, 30].

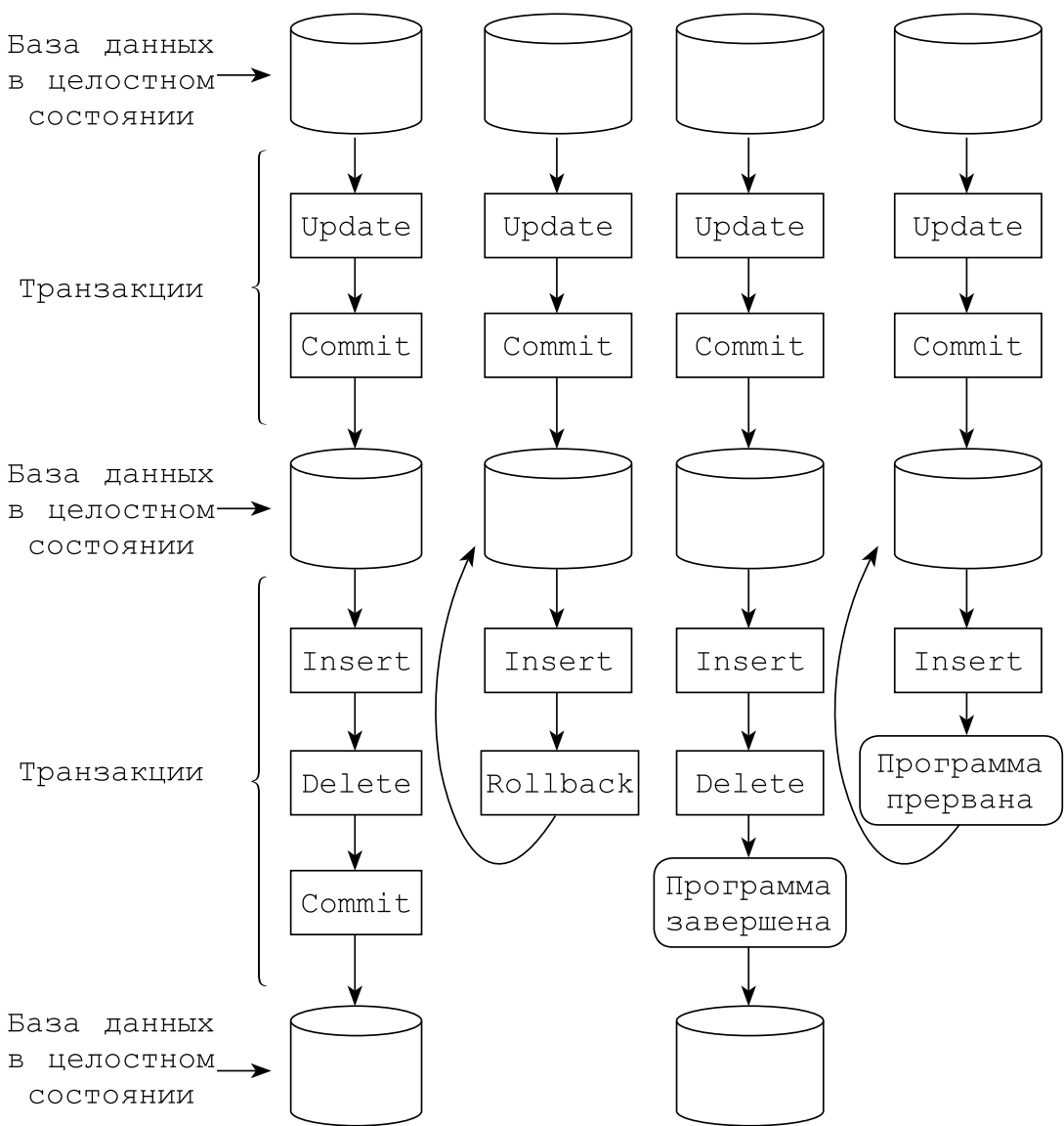


Рис. 8.1. Выполненные и отмененные транзакции

В СУБД Firebird существует возможность задавать *пользовательские точки сохранения* (альтернативное имя — *вложенные транзакции*) и затем откатывать транзакцию не полностью, а частично — до этих точек [5]. Место расположения точки сохранения в транзакции указывает оператор SAVEPOINT, который имеет синтаксис

```
SAVEPOINT имя_точки_сохранения; .
```

Правило составления имя_точки_сохранения такое же, как и для любых других идентификаторов языка SQL (см. параграф 2.7 (ч. 1)).

В транзакции может быть определено множество точек сохранения, причем если в одной транзакции повторно определяется точка с уже использовавшимся именем, то она перекрывает существовавшую точку, связанную с этим именем.

Возврат на точку сохранения начинается с отмены всей работы, выполненной после создания этой точки сохранения. Указанная точка сохранения и все предшествующие ей сохраняются, а те точки, которые были созданы после указанной точки сохранения, пропадают. Еще одна особенность отката до контрольной точки заключается в том, что транзакция не завершается. Иными словами, оставшиеся изменения, если требуется, нужно зафиксировать.

Рассмотрим пример использования точек сохранения в Firebird:

```
CREATE TABLE Test (ID INTEGER);
COMMIT;
INSERT INTO Test VALUES (1);
COMMIT;
INSERT INTO Test VALUES (2);
SAVEPOINT y;
DELETE FROM Test;
SELECT * FROM Test; -- не вернет ничего
ROLLBACK TO y;
SELECT * FROM Test; -- вернет две строки
ROLLBACK;
SELECT * FROM Test; -- вернет одну строку
```

Механизм реализации точек сохранения на сервере может требовать значительного количества ресурсов, особенно если одни и те же строки изменяются многократно в течение одной транзакции. Ресурсы уже ненужных точек сохранения могут быть освобождены при использовании оператора `RELEASE SAVEPOINTS`, имеющего синтаксис

```
RELEASE SAVEPOINTS имя_точки_сохранения [ONLY];
```

Без ключевого слова `ONLY` указанная точка сохранения и все точки сохранения, которые были созданы после нее, будут освобождены и потеряны. `ONLY` используется для освобождения только указанной точки сохранения.

Точки сохранения оказываются полезными в интерактивных программах, так как они позволяют выделять и именовать промежуточные шаги программы [26]. Это помогает гибко управлять сложными программами. Например, можно проставить точки сохранения в длинной цепочке обновлений. При этом, обнаружив ошибки, можно будет произвести откат к нужному месту и тем самым не выполнять повторно все обновления, как если бы был произведен откат всей транзакции.

Точки сохранения также оказываются полезными при написании прикладных программ. Если программа содержит подпрограммы, то можно проставить точки сохранения перед обращением к каждой из подпрограмм. Если подпрограмма завершилась аварийно, можно восстановить БД на момент до выполнения этой подпрограммы.

Система управления БД Oracle, как и Firebird, поддерживает для управления транзакциями операторы COMMIT, ROLLBACK и SAVEPOINT [1].

В Oracle, как и в Firebird, нет явного оператора, чтобы начать транзакцию, и нет автоматического завершения транзакции. Транзакция автоматически начинается с первого оператора, который начал изменения, т. е. установил блокировку, а заканчивается явным оператором окончания транзакции. При выполнении DDL-запросов, например при использовании CREATE TABLE для добавления новой таблицы в БД, Oracle сразу изменяет БД, что видно любому пользователю, обращающемуся к таблице. Для DML-запросов (INSERT, UPDATE, DELETE) по умолчанию это не так. Например, если выполнен запрос DELETE и не выполнялся выход из системы, то можно восстановить удаляемую запись. Если же был выполнен выход из системы после удаления записи, то восстановление невозможно, поскольку Oracle при выходе выполняет неявную команду фиксации транзакций. Изменить принцип сохранения изменений, вызываемых DML-запросами, можно путем установки специальной переменной AUTOCOMMIT, которая отвечает за немедленное сохранение изменений в БД сразу же после выполнения запросов.

Язык управления транзакциями TCL в MS SQL Server включает в себя операторы, осуществляющие запуск, подтверждение, откат или создание точки сохранения транзакции [4]:

- операторы старта обычной или распределенной транзакции: BEGIN TRANSACTION и BEGIN DISTRIBUTED TRANSACTION;
- операторы подтверждения транзакции: COMMIT TRANSACTION, COMMIT WORK;
- операторы отката транзакции: ROLLBACK TRANSACTION, ROLLBACK WORK;
- оператор создания точки сохранения SAVE TRANSACTION.

В MS SQL Server существует два типа транзакций: неявные и явные [27]. *Неявная транзакция* задает любой отдельный запрос INSERT, UPDATE или DELETE как единицу транзакции. *Явная транзакция* обычно представляет собой группу запросов языка Transact-SQL, начало и конец которой обозначаются операторами BEGIN TRANSACTION, COMMIT и ROLLBACK.

Оператор BEGIN TRANSACTION используется для запуска явной транзакции и имеет следующий синтаксис:

```
BEGIN TRANSACTION [ { имя_транзакции | @имя_переменной }  
[ WITH MARK ['строка']] ];
```

Имя транзакции можно использовать в операторе только в самой внешней паре вложенных операторов BEGIN TRANSACTION/COMMIT или BEGIN TRANSACTION/ROLLBACK. В параметре @имя_переменной указывается имя определяемой пользователем переменной, содержащей действительное имя транзакции.

Опция WITH MARK указывает, что транзакция должна быть отмечена в журнале. Аргумент строка описывает эту отметку. В случае использования опции WITH MARK необходимо указывать имя транзакции.

Оператор BEGIN DISTRIBUTED TRANSACTION используется для запуска распределенной транзакции, которая управляется Microsoft Distributed Transaction Coordinator (MS DTC — координатором распределенных транзакций Microsoft). *Распределенная транзакция* — это транзакция, которая используется на нескольких БД и на нескольких серверах.

Оператор COMMIT WORK успешно завершает транзакцию, запущенную инструкцией BEGIN TRANSACTION. Язык Transact-SQL также поддерживает оператор COMMIT TRANSACTION, который функционально равнозначен оператору COMMIT WORK с той разницей, что он принимает определяемое пользователем имя транзакции. Оператор COMMIT TRANSACTION является расширением языка Transact-SQL, соответствующим стандарту SQL.

Также поддерживается оператор ROLLBACK TRANSACTION, который функционально равнозначен оператору ROLLBACK WORK с той разницей, что он принимает определяемое пользователем имя транзакции.

Оператор SAVE TRANSACTION устанавливает точку сохранения внутри транзакции. Аналогичен рассмотренному выше для Firebird оператору SAVEPOINT.

8.3. Параллелизм

Одной из основных задач многопользовательских СУБД является организация одновременного доступа к одним и тем же данным множеству пользователей с помощью механизма транзакций. Таким образом, когда две или более задач выполняются над одной и той же БД в одно и то же время, говорят о *параллелизме*. Основными проблемами, возникающими при параллельной обработке транзакций, являются следующие [5, 8]:

- потерянные или «скрытые» изменения (*lost updates*);
- зависимость от незафиксированных данных («грязное» чтение — *dirty read*);

- несогласованный анализ (неповторяемое чтение, невозпроизводимое чтение — *non-reproducible-reads*);
- чтение фантомов (фантомные строки — *phantom rows*);
- перекрывающиеся транзакции.

Потерянные изменения появляются в ситуации, когда два и более пользователя (или программы) читают одни и те же данные из БД, вносят в них какие-либо изменения и затем пытаются одновременно записать результат на прежнем месте. В этом случае ни у одной из транзакций нет сведений о действиях, выполненных другими транзакциями. В БД могут быть сохранены изменения, выполненные только одной программой, — любые другие изменения будут потеряны.

Зависимость от незафиксированных данных возникает, когда вторая транзакция (пользователь) выбирает строку, которую в это время обновляет первая транзакция (другой пользователь), без гарантии того, что эти изменения являются окончательными. Затем первая транзакция откатывается оператором ROLLBACK. Таким образом, вторая транзакция читает еще не зафиксированные данные, которые могут быть изменены первой транзакцией. Следует отметить, что Firebird не допускает «грязного» чтения.

Несогласованный анализ возникает тогда, когда один пользователь выбирает строки (например, получает какой-то итоговый отчет), а в это время другие пользователи изменяют или удаляют эти данные. Несогласованный анализ напоминает неподтвержденную зависимость тем, что первая транзакция изменяет данные, которые читает вторая транзакция. Однако при несогласованном анализе первая транзакция подтверждает изменение этих данных. Кроме того, при несогласованном анализе происходит многократное чтение второй транзакцией одной и той же строки с разными данными.

Чтение фантомов возникает, когда последующий набор строк, прочитанный транзакцией, отличается от набора, который был прочитан в начале работы транзакции. Фантомные строки появляются, если при последующем чтении появляются новые добавленные строки и (или) исчезают удаленные строки, которые были подтверждены с момента первого чтения.

Перекрывающиеся транзакции могут возникать, когда изменения строки одним пользователем влияют на данные в других строках в той же таблице или в других таблицах, к которым обращаются другие пользователи.

Для решения вышеперечисленных проблем необходим определенный механизм обработки транзакций, позволяющий устранить подобные проблемы. Такой механизм существует и опирается на следующие правила [17, 29].

1. В процессе выполнения транзакции пользователь (или программа) «видит» только согласованные состояния БД. Пользователь

(или программа) никогда не может получить доступ к незафиксированным обновлениям в данных, достигнутым в результате действий другого пользователя (программы).

2. Если две транзакции А и В выполняются параллельно, то СУБД полагает, что результат будет такой же, как если бы:

- транзакция А выполнялась первой, за ней была бы выполнена транзакция В;
- транзакция В выполнялась бы первой, за ней была бы выполнена транзакция А.

Администратор транзакций гарантирует, что каждый пользователь (программа), обращающийся к БД, работает с ней так, как будто не существует других пользователей (программ), одновременно с ним обращающихся к тем же данным.

Для практической реализации целостности и согласованности данных БД различные СУБД используют различные механизмы обработки транзакций.

Многие СУБД (например, MS SQL Server) используют *механизм блокировок* [7, 29]. Если для выполнения транзакции нужно, чтобы некоторый объект не изменялся непредсказуемо и без ведома этой транзакции, то такой объект блокируется.

Механизм блокировок разрешает проблемы, связанные с доступом нескольких пользователей (программ) к одним и тем же данным. Однако его применение связано с существенным замедлением обработки транзакций, вызванным необходимостью ожидания освобождения данных, захваченных конкурирующей транзакцией.

Возможно ускорение обработки путем локализации фрагментов данных, захватываемых транзакцией.

Система управления БД может блокировать:

- всю базу данных целиком;
- таблицу БД;
- часть таблицы;
- отдельную строку.

Это называется **уровнями блокировки**. Современные СУБД в основном используют блокировки на уровне частей таблиц (страниц) и (или) на уровне записей.

При блокировке на уровне страниц СУБД захватывает для выполнения транзакции некоторый фрагмент таблицы, запрещая доступ к нему (на время выполнения транзакции) конкурирующим транзакциям. Последние, впрочем, могут захватить другие страницы той же таблицы. Поскольку размер страниц обычно невелик (1—8 Кбайт), то время ожидания транзакций, конкурирующих за доступ к страницам таблицы, оказывается приемлемым даже для режима оперативного доступа к БД.

Если СУБД имеет возможность блокировать для выполнения транзакции отдельные строки таблицы, то скорость обработки

транзакции существенно повышается. Блокировка на уровне записей (строк) позволяет добиться максимальной производительности за счет того, что захватываемый объект (запись) является минимальной структурной единицей БД. Блокировка на уровне записей (row-level) реализована в Firebird.

Теоретически блокировка на уровне элементов данных (захват конкретного поля строки) позволит добиться еще большей производительности.

Для пояснения действия механизма блокировок на уровне записей, реализованного в СУБД Firebird, воспользуемся рис. 8.2. На рисунке представлены таблицы А и В учебной БД (PaySumma, NachislSumma), к которым возможен доступ в процессе выполнения двух транзакций. Сначала при выполнении транзакции А происходит обновление первой строки таблицы PaySumma. Система управления БД блокирует данную запись до тех пор, пока транзакция А не будет зафиксирована или отменена.

Транзакция В выполняется параллельно и блокирует на момент своего выполнения первую запись таблицы NachislSumma. Затем в контексте транзакции А происходит выборка из таблицы NachislSumma. Такая выборка возможна (транзакцией В блокирована первая строка таблицы только для обновления, но не для чтения), однако транзакция А не увидит изменений, сделанных неподтвержденным обновлением в транзакции В. Затем в процессе выполнения транзакции В делается попытка доступа к блокированной первой строке таблицы PaySumma. В результате обработка транзакции В приостанавливается и возобновляется только после того, как транзакция А завершается и освобождает блокированную ею строку таблицы PaySumma. Вместе с тем попытка обновления второй строки таблицы PaySumma в процессе выполнения транзакции В будет успешной, так как заблокирована не вся таблица PaySumma, а лишь первая строка.

Помимо уровней блокировки, выделяют также тип блокировки или схему блокировки. Конкурирующие транзакции могут захватывать данные, в то же время разрешая доступ к этим данным другим транзакциям, но только для чтения. Кроме того, транзакции могут блокировать данные, не допуская захвата тех же данных другими транзакциями, в том числе и только для чтения.

Система управления БД Firebird относится к СУБД с многоверсионной архитектурой (MGA — *MultiGenerational Architecture*) [5]. Система управления БД с такой архитектурной моделью управления состоянием БД для обработки транзакций используют *механизм версионности записей* БД — позволяют сохранять в БД более одной версии записи одновременно. Часто такие СУБД называют «версионниками», в то время как СУБД, использующие только механизм блокировок, — «блокировочниками».

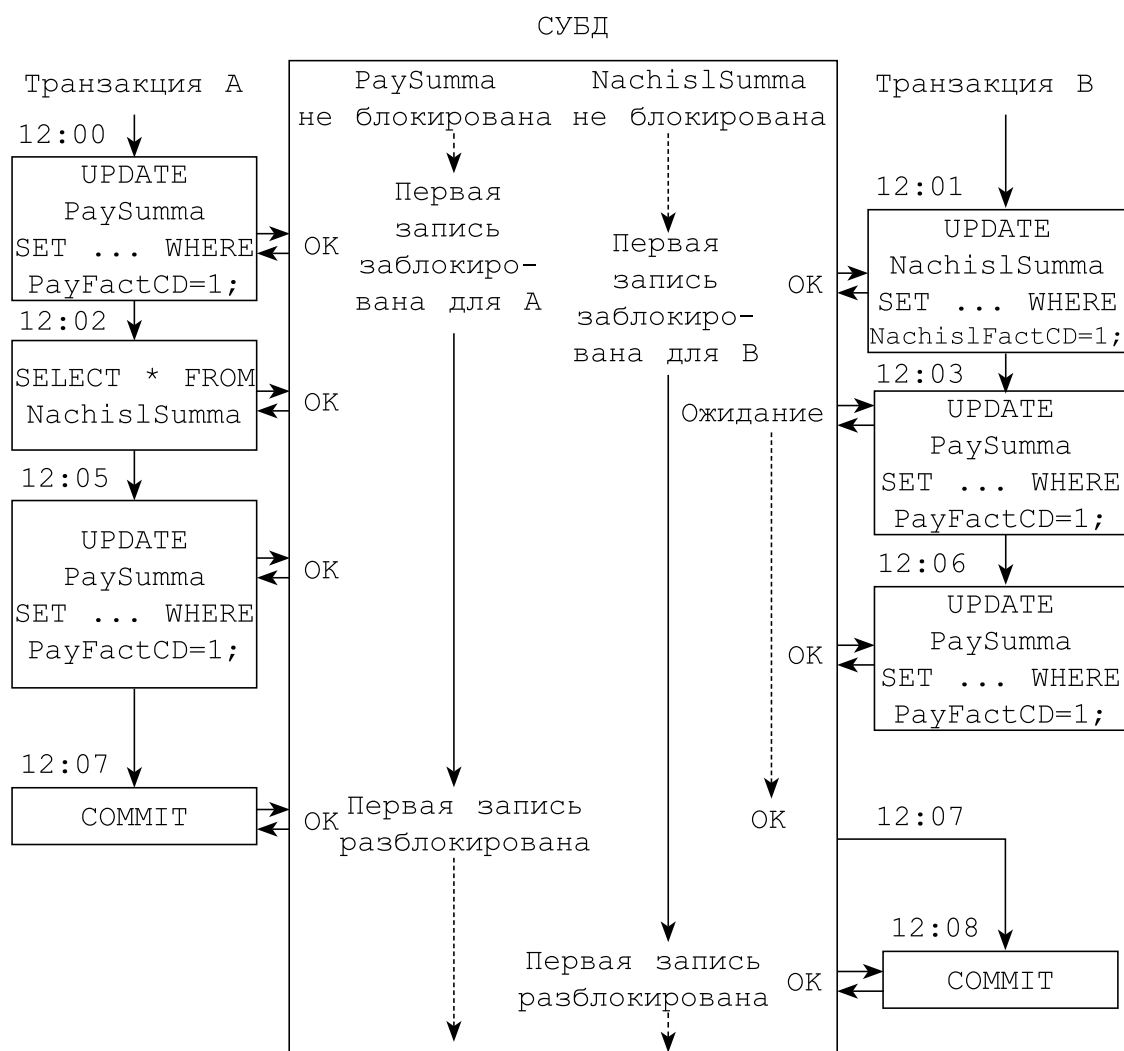


Рис. 8.2. Использование блокировок на уровне записей

В модели MGA каждая пользовательская задача записывает свои версии записей на диск и каждая строка, сохраняемая в БД, содержит уникальный идентификатор той транзакции, которая ее сохранила.

В момент изменения записи в одной из транзакций новая версия записи (или удаленная запись) недоступна другим задачам пользователей. Только самая последняя подтвержденная версия записи является видимой за пределами пользовательской задачи, которая успешно сохранила новую версию, и эта запись продолжает оставаться видимой для других задач. Другие задачи будут в курсе того, что что-то произошло с этой записью, поскольку они будут блокированы от изменения или удаления этой записи, пока новая версия не станет «официальной» после подтверждения изменений.

Если другая транзакция посылает изменение строки и строка не блокирована другой транзакцией, то сервер записывает на диск новую версию этой строки с новым идентификатором транзакции и преобразует образ старой версии в ссылку (называемую дельтой)

на эту новую версию. Теперь сервер содержит два «поколения» одной и той же строки.

Поскольку Firebird является «версионником», то у него нет никаких проблем, свойственных «блокировочнику» [31]. Операция чтения записи никогда не приводит к зависаниям, даже если кто-либо параллельно осуществляет модификацию той же самой записи. Читающая транзакция никогда не блокирует транзакцию на изменение (и наоборот). Существует лишь одно ограничение: несколько пользователей одновременно не могут изменять одну и ту же запись.

Если все-таки иногда возникает необходимость временно установить блокировку на ту или иную запись, в Firebird это решается приемом, известным как «холостое обновление», либо с помощью конструкции `SELECT FOR UPDATE WITH LOCK`. В обоих случаях создается новая версия записи, и именно она блокируется. При этом операция чтения старой версии записи не блокируется.

Таким образом, СУБД Firebird использует смешанный способ обработки параллельных транзакций — создание версий записей и блокировки на уровне строк таблиц.

Контрольные вопросы и задания

1. Что может быть причиной разрушения или потери данных в БД?
2. Что такое транзакция? Какими ACID-свойствами характеризуются транзакции? В чем состоят преимущества использования транзакций?
3. Что такое фиксация транзакции? Как осуществляется фиксация транзакции на языке SQL?
4. Что такое откат транзакции? Как осуществляется откат транзакции на языке SQL?
5. Как можно применять пользовательские точки сохранения при работе с транзакциями?
6. Какие основные проблемы возникают при параллельной обработке транзакций?
7. Опишите механизм обработки транзакций, основанный на блокировках.
8. Опишите механизм обработки транзакций, основанный на версионности записей.

Лабораторный практикум

Лабораторный практикум содержит восемь работ по всем разделам настоящего учебника. Лабораторные задания и сценарии выполнения лабораторных работ № 2—8 для каждой СУБД одинаковые. Отличия имеются только в работе № 1, которые заключаются в использовании различных клиентских приложений и скриптов для создания учебной БД. В настоящем учебнике приводятся методические указания для выполнения работы № 1 только для СУБД Firebird¹.

Каждая лабораторная работа предполагает выполнение в интерактивном режиме общей части и индивидуального задания.

Защита лабораторных работ производится под контролем информационной системы [32-34]. Для защиты лабораторных работ № 1, 6, 7 и 8 необходимо результативно ответить на определенное число контрольных вопросов и заданий (в виде традиционного тестирования), а для защиты работ № 2, 3, 4 и 5 — построить и ввести SQL-запросы согласно определенному числу предлагаемых заданий различной сложности.

Описание лабораторной установки. Компьютер с локально установленными сервером БД (Firebird SQL Server, MS SQL Server или Oracle Database) и соответствующим клиентским приложением (IBExpert, dbForge Studio for SQL Server или dbForge Studio for Oracle). Адреса для их копирования, а также инструкции по установке и настройке приведены на странице настоящего учебника на сайте <https://rgrty.ru>.

Лабораторная работа № 1. IBExpert — инструмент разработчика и администратора баз данных Firebird. Учебная база данных

Цель работы. Знакомство с возможностями и приобретение практических навыков работы с интерактивной утилитой IBExpert как инструментом разработчика и администратора баз данных СУБД Firebird. Создание учебной БД.

¹ На сайте <https://rgrty.ru> можно также выполнить лабораторные практикумы для СУБД MS SQL и Oracle, а на сайте <https://rgrty.ru/sqltest/> — пройти тестирование.

Лабораторное задание. Изучить теоретическую часть, изложенную в гл. 1 и 2 настоящего учебника (ч. 1). Ознакомиться с предметной областью (учебной БД) по описанию в приложении А. При необходимости установить на используемый компьютер Firebird SQL Server (SuperServer) версии 3.0 и утилиту IBExpert. Создать учебную БД и заполнить ее таблицы данными, взятыми из описания ее одноименных таблиц. В качестве примеров выборки данных выполнить запросы к таблицам БД. Выполнить резервное копирование и восстановление учебной БД. Составить отчет и защитить лабораторную работу.

Порядок выполнения работы

1. Создать средствами используемой операционной системы на физическом диске **C:** каталоги **GGGG\SQL**, где **GGGG** соответствует номеру группы.

2. Загрузить утилиту IBExpert, предварительно убедившись, что сервер СУБД Firebird загружен.

2.1. Ознакомиться с правилами работы и структурой меню, а также со всеми доступными пунктами меню утилиты.

2.2. Создать, используя пункт главного меню **База данных/Создать базу** утилиты IBExpert, в каталоге **C:\GGGG\SQL** файл рабочей БД с именем **NN.fdb**, где **NN** соответствует номеру варианта.

При этом сервер выбрать **«Remote, TCP/IP»** с именем **«localhost»**, имя пользователя и пароль задать как для администратора БД, выбрать диалект БД — **3**, установить размер страницы — **16384** (это размер страницы по умолчанию), а кодировку — **WIN1251**.

3. Зарегистрировать созданную БД, выбрав версию сервера Firebird 3.0, задав произвольное описание (имя) БД, а значения всех остальных параметров БД — установленными по умолчанию. Подключиться к созданной БД и ознакомиться со всеми вновь доступными пунктами меню утилиты.

4. Вызвать SQL-редактор (**Инструменты/SQL Редактор**). Создать четыре домена, последовательно вводя и выполняя запросы:

```
CREATE DOMAIN Money AS NUMERIC(15,2);
CREATE DOMAIN TMonth AS SMALLINT CHECK (VALUE BETWEEN 1 AND 12);
CREATE DOMAIN PKfield AS INTEGER;
CREATE DOMAIN TYear AS SMALLINT CHECK (VALUE BETWEEN 1990 AND 2100);
```

5. Создать в учебной БД таблицу **Services**, выполнив запрос

```
CREATE TABLE Services (ServiceCD PKFIELD,
    ServiceNM VARCHAR(30), PRIMARY KEY (ServiceCD));
```

Подтвердить транзакцию и убедиться в наличии создаваемых объектов БД (**Database Explorer**).

6. Вставить в таблицу услуг (Services) все строки с данными, взятыми из описания одноименной таблицы учебной БД, используя аналогию со следующим запросом INSERT, который применяется для вставки первой строки в таблицу услуг:

```
INSERT INTO Services (ServiceCD, ServiceNM)  
VALUES (1, 'Газоснабжение');
```

7. Вывести содержимое таблицы Services с помощью запроса

```
SELECT * FROM Services;
```

и убедиться в правильности введенных данных.

8. В случае правильного ввода данных зафиксировать текущую транзакцию. Если данные, выведенные запросом п. 7, отличаются от данных соответствующей таблицы, то следует выполнить откат текущей транзакции и повторить п. 6 и 7.

9. Сформировать скрипт для создания и заполнения всех остальных таблиц учебной БД. Для этого в редактор скриптов (**Инструменты/Редактор скриптов**) загрузить файл со скриптом, находящимся в приложении В, и удалить в нем запросы, выполненные в п. 2—6. Выполнить полученный скрипт, предварительно добавив в него оператор подключения к рабочей БД (CONNECT) или установив галочку **Использовать текущее соединение**. Снимок экрана 1-й страницы сформированного скрипта включить в отчет.

10. Просмотреть на экране всю информацию о созданных объектах БД. По результатам анализа в случае успешного создания БД зафиксировать изменения, в противном случае — отменить изменения и повторить п. 9. Выполнить в SQL-редакторе следующий запрос, который извлекает из таблицы NachislSumma все строки о начислениях абоненту с ФИО Тимошкина Н. Г. (номер лицевого счета '080270'):

```
SELECT * FROM NachislSumma WHERE AccountCD = '080270';
```

11. Выполнить запрос, который извлекает из таблицы PaySumma всю информацию об оплатах услуг, произведенных абонентами за период позже 2016 г.:

```
SELECT * FROM PaySumma WHERE PayYear > 2016; .
```

12. Выполнить запрос, который возвращает сумму значений всех оплат, произведенных каждым абонентом:

```
SELECT AccountCD, SUM(PaySum) AS Summa_Abonent  
FROM PaySumma  
GROUP BY AccountCD;
```

13. Проверить действие всех ограничений ссылочной целостности (запрещающее, каскадное, обновления) путем внесения соответствующих изменений в данные таблиц учебной БД, заданных вариантом индивидуального задания.

14. Проверить блокировку ввода для таблиц учебной БД, заданных вариантом индивидуального задания.

15. Проверить действие ограничений доменной целостности. Для этого в соответствующих столбцах таблиц учебной БД, заданных вариантом индивидуального задания, изменить значения на недопустимые.

16. Отключиться от БД без фиксации изменений. В отчете зафиксировать снимок экрана с размером файла БД (в байтах) и временем создания. Создать сжатую резервную копию (backup) БД (**Службы/Резервирование БД**). В отчете зафиксировать снимок экрана с размером (в байтах) и временем создания файла сжатой резервной копии БД.

17. Восстановить БД из сжатой резервной копии (**Службы/Восстановление БД**). В отчете зафиксировать снимок экрана с размером (в байтах) и временем создания файла восстановленной БД. Сравнить его с размерами файлов первоначальной БД и ее сжатой резервной копии. Скопировать сжатую резервную копию БД на внешний носитель данных для использования ее в последующих лабораторных работах.

18. Составить отчет. В отчете представить результаты выполнения п. 2.2, 7, 9—17.

19. Защитить лабораторную работу, положительно ответив под контролем информационной системы [32] на заданное количество тестовых вопросов по теме настоящей работы.

Варианты индивидуальных заданий

Вариант 1

Street с Abonent.

Вариант 2

Executor с Request.

Вариант 3

Disrepair с Request.

Вариант 4

Abonent с Request.

Вариант 5

Abonent с Paysumma.

Вариант 6

Abonent с Nachislsomma.

Вариант 7

Services с Paysumma.

Вариант 8

Services с Nachislsomma.

Вариант 9

Executor и Disrepair с Request.

Вариант 10

Abonent и Disrepair с Request.

Вариант 11

Abonent и Services с Nachislsomma.

Вариант 12

Abonent и Services с Paysumma.

Вариант 13

Abonent и Executor с Request.

Вариант 14

Services с Nachislsomma и Paysumma.

Вариант 15

Abonent с Nachislsomma и Paysumma.

Лабораторная работа № 2. Программирование однотабличных запросов выборки данных

Цель работы. Развитие умения формулировать задания на получение информации из БД; приобретение практических навыков построения запросов на выборку данных из одной таблицы.

Лабораторное задание. Изучить теоретическую часть, изложенную в параграфах 3.1 и 3.2 (ч. 1). В ходе самостоятельной подготовки сформулировать в терминах предметной области задания на получение информации, имеющей определенное прикладное значение, из таблиц, заданных вариантом в лабораторной работе № 1:

- с каждым типом возвращаемого элемента в секции SELECT;
- с каждым видом проверки в условии поиска секции WHERE;
- с группировкой, в том числе и с HAVING;
- с сортировкой;
- с использованием одной встроенной функции из каждой группы.

Построить и выполнить соответствующие однотабличные запросы на выборку. Оформить отчет. Защитить лабораторную работу.

Порядок выполнения работы

1. Убедившись, что сервер СУБД загружен, загрузить среду разработки.

2. Восстановить учебную БД из резервной копии, созданной в лабораторной работе № 1. При необходимости создать новую БД можно с помощью скрипта, используемого в лабораторной работе № 1.

3. Подключиться к учебной БД.

4. Построить согласно сформулированным заданиям и выполнить с помощью SQL-редактора среды разработки селективные запросы к соответствующим таблицам учебной БД.

4.1. С каждым типом возвращаемого элемента в секции SELECT.

4.2. С каждым видом проверки в условии поиска секции WHERE.

4.3. С группировкой, в том числе и с HAVING.

4.4. С сортировкой.

4.5. С использованием в любой из секций запроса встроенной функции из каждой группы.

Убедиться, что результат каждого запроса соответствует условиям соответствующего задания. Если получился результат, отличный от ожидаемого, то следует исправить запрос и выполнить его снова.

5. Составить отчет, в котором по каждому заданию п. 4 представить формулировку, запрос, результаты его выполнения и проверки правильности решения. Все задания должны быть пронумерованы, например 4.1.1, 4.1.2, ..., 4.2.1, 4.2.2, ..., 4.5.1, 4.5.2 и т. д. Запросы представить в текстовом формате.

6. Защитить лабораторную работу, построив и результативно выполнив под контролем информационной системы [32] заданное количество запросов к учебной БД.

Лабораторная работа № 3. Программирование многотабличных и вложенных запросов выборки данных

Цель работы. Развитие умения формулировать задания на получение информации из БД; приобретение практических навыков построения многотабличных и вложенных запросов на выборку данных.

Лабораторное задание. Изучить теоретическую часть, изложенную в параграфе 3.3 (ч. 1). В ходе самостоятельной подготовки сформулировать в терминах предметной области задания на получение информации, имеющей определенное прикладное значение, из таблиц, заданных вариантом в лабораторной работе № 1:

- с неявным соединением;
- с каждым видом явного соединения;
- с каждым типом независимых подзапросов;
- с каждым типом связанных подзапросов;
- с каждым из предикатов ANY, ALL, SINGULAR или EXISTS;
- с объединением запросов.

Построить и выполнить соответствующие запросы к таблицам учебной БД на выборку данных. Защитить лабораторную работу.

Порядок выполнения работы

1. Подключиться к учебной БД.
2. Построить и выполнить с помощью SQL-редактора среды разработки многотабличные и вложенные запросы к соответствующим таблицам учебной БД.

- 2.1. С неявным соединением, в том числе и условием поиска.
- 2.2. С каждым видом явного соединения.
- 2.3. С каждым типом независимых подзапросов.
- 2.4. С каждым типом связанных подзапросов.
- 2.5. С каждым из предикатов: ANY, ALL, SINGULAR или EXISTS.
- 2.6. С объединением запросов.

Убедиться, что результат запроса соответствует условиям задания. Если получился результат, отличный от ожидаемого, то следует исправить запрос и выполнить его снова.

3. Составить отчет, в котором по каждому заданию п. 2 представить формулировку, запрос, результаты его выполнения и проверки правильности задания. Все задания должны быть пронумерованы, например 2.1.1, 2.1.2, ..., 2.2.1, 2.2.2, ..., 2.6. Запросы представить в текстовом формате.

4. Защитить лабораторную работу, построив и результативно выполнив под контролем информационной системы [32] заданное количество запросов к учебной БД.

Лабораторная работа № 4. Программирование запросов определения данных

Цель работы. Приобретение практических навыков построения запросов на создание и изменение структуры объектов БД.

Лабораторное задание. Изучить теоретическую часть, изложенную в гл. 4 (ч. 1). Построить и выполнить запросы DDL к таблицам учебной БД, заданным вариантом в лабораторной работе № 1. Защитить лабораторную работу.

Порядок выполнения работы

1. Подключиться к учебной БД. Выполнить с объектами учебной БД нижеприведенные действия, заданные вариантом индивидуального задания.

2. Проанализировать запрос CREATE TABLE, использованный для создания дочерней таблицы учебной БД. Результатом анализа должна быть информация о первичных и внешних ключах таблицы, которую необходимо занести в отчет.

3. Определить свойства дочерней таблицы учебной БД, гарантирующие целостность данных (целостность сущностей, целостность ссылок, доменную целостность и целостность, определяемую пользователем).

4. Создать таблицу — справочник марок газового оборудования, заданного вариантом типа, состоящую из двух столбцов (один из которых должен быть определен как первичный ключ). Эта таблица должна быть родительской по отношению к таблице Abonent, т. е. иметь первичный ключ, на который ссылается соответствующий внешний ключ таблицы Abonent. Определение первичного ключа создаваемой таблицы и внешнего ключа таблицы Abonent реализовать таким образом, чтобы обеспечить условия ссылочной целостности, заданные вариантом индивидуального задания. При этом создаваемым ограничениям должно быть дано произвольное имя. Проверить работу созданных правил изменения внешнего ключа. Удалить созданную таблицу.

5. Создать индекс по символьному полю родительской таблицы, отсортированный в обратном алфавитном порядке.

6. Добавить к выбранным столбцам двух таблиц учебной БД соответственно два ограничения CHECK, запрещающие хранение ошибочных или не имеющих смысла значений. При создании ограничений каждому из них присвоить уникальное имя. Убедиться, что ограничения добавлены в таблицу, и проверить их действие. Изме-

нить одно из ограничений. Убедиться, что ограничение изменено, и проверить его действие. Удалить добавленные ограничения.

7. Добавить в дочернюю таблицу учебной БД столбец с вычисляемыми значениями на основе значений из существующих столбцов. Убедиться, что изменения добавлены в таблицу и проверить их действие. Удалить вычисляемый столбец.

8. Установить выбранному столбцу таблицы учебной БД значение по умолчанию. Убедиться, что изменения добавлены в таблицу, и проверить их действие. Изменить значение по умолчанию и проверить его действие. Удалить добавленное значение по умолчанию.

9. Создать комментарий для одного объекта учебной БД, выбранного самостоятельно. Изменить текст комментария. Удалить комментарий.

10. Создать к таблицам учебной БД представление каждого типа и получить данные, возвращаемые ими. Исследовать работу запросов RECREATE VIEW, ALTER VIEW и CREATE OR ALTER VIEW. Удалить созданные представления.

11. Оформить отчет. В отчете представить результаты анализа структуры учебной БД, SQL-запросы, использованные при выполнении всех заданий, а также результаты их выполнения.

12. Защитить лабораторную работу, положительно ответив под контролем информационной системы [32] на заданное количество тестовых вопросов по теме настоящей лабораторной работы.

Варианты индивидуальных заданий

Вариант 1

Pg (справочник марок газовых плит); D:SN, U:C.

Вариант 2

Vk (справочник марок газовых водонагревателей); D:C, U:SN.

Вариант 3

Op (справочник марок газовых отопительных приборов); D:SD, U:SD.

Вариант 4

Sh (справочник марок газовых счетчиков); D:C, U:C.

Вариант 5

Pg (справочник марок газовых плит); D:R, U:R.

Вариант 6

Vk (справочник марок газовых водонагревателей); D:SN, U:SN.

Вариант 7

Op (справочник марок газовых отопительных приборов); D:R, U:C.

Вариант 8

Sh (справочник марок газовых счетчиков); D:C, U:SN.

Вариант 9

Pg (справочник марок газовых плит); D:R, U:SN.

Вариант 10

Vk (справочник марок газовых водонагревателей); D:C, U:SD.

Вариант 11

Op (справочник марок газовых отопительных приборов); D:SD, U:SN.

Вариант 12

Sh (справочник марок газовых счетчиков); D:SN, U:SD.

Вариант 13

Pg (справочник марок газовых плит); D:C, U:SD.

Вариант 14

Vk (справочник марок газовых водонагревателей); D:R, U:SD.

Вариант 15

Op (справочник марок газовых отопительных приборов); D:R, U:SN.

Лабораторная работа № 5. Программирование запросов модификации данных

Цель работы. Приобретение практических навыков построения запросов на модификацию данных.

Лабораторное задание. Изучить теоретическую часть, изложенную в гл. 5. Построить и выполнить запросы DML к таблицам учебной БД. Защитить лабораторную работу.

Порядок выполнения работы

1. Подключиться к учебной БД.
2. Создать таблицу Pattern, используя запрос

```
CREATE TABLE Pattern  
(Fint INTEGER,  
Fchar30 VARCHAR(30),  
Fchar40 VARCHAR (40),  
Fchar8 VARCHAR (8),  
Fnumeric NUMERIC(18,4),  
Fdate DATE);
```

Таблица Pattern предназначена для временного хранения данных разного типа, извлекаемых из таблиц учебной БД. Если некоторые столбцы таблицы Pattern не используются, то их значения будут установлены в NULL.

3. Поместить в таблицу Pattern данные из таблиц учебной БД, используя многострочный запрос INSERT, в соответствии с заданиями, приведенными ниже. Если при выполнении заданий возникнет необходимость отмены изменений, то следует отменить текущую транзакцию.

3.1. Информация о средних значениях начислений, приходящихся на каждый месяц года, выбранного самостоятельно. При этом столбец Fchar30 должен содержать номер месяца и год начисления, разделенные десятичной точкой, столбец Fnumeric — среднее значение начислений.

3.2. Информация об абонентах, проживающих на улице, выбранной самостоятельно. При этом столбец Fchar30 должен содержать ФИО абонента, а столбец Fchar40 — номер дома и номер квартиры, в которой проживает абонент.

3.3. Информацию о заявках и сделавших их абонентах, проживающих на улице п. 3.2. При этом столбец Fnumeric должен содержать номер заявки, столбец Fdate — дату подачи заявки, столбец Fchar30 — ФИО абонента, сделавшего заявку, столбец Fchar8 — номер лицевого счета абонента, а столбец Fchar40 — название улицы, на которой проживает данный абонент.

4. С помощью однострочного запроса INSERT поместить в таблицу ремонтных заявок новую заявку от абонента, выбранного самостоятельно. Код заявки должен автоматически вычисляться как следующий по порядку. Исполнителем этой заявки назначить мастера, выбранного самостоятельно. В качестве даты подачи заявки задать дату, равную текущей, уменьшенной на один месяц. В качестве неисправности указать любую из таблицы неисправностей. Дата выполнения заявки не должна быть определена.

5. Установить для новой заявки дату выполнения равной текущей дате. При этом код заявки считать неизвестным.

6. Добавить в таблицу регистрации фактов оплаты за услуги новый факт оплаты, принятой от абонента, выбранного самостоятельно. Код факта оплаты должен автоматически вычисляться как следующий по порядку, а значения остальных параметров оплаты задать произвольно.

7. Для строки, добавленной в п. 6 (считать код факта оплаты неизвестным), изменить дату оплаты на определенную как 10-е число месяца, следующего за оплачиваемым месяцем.

8. Отменить все сделанные изменения в БД.

9. Построить и выполнить запросы в соответствии с вариантом индивидуального задания. Каждое задание должно быть выполнено с помощью одного запроса. После выполнения каждого запроса и просмотра изменений, которые им произведены, необходимо отменять текущую транзакцию. После выполнения запросов, удаляющих строки из таблиц, необходимо просматривать их содержимое, чтобы проследить за изменениями, которые обусловлены наличием связей «родитель — потомок» (первичный — внешний ключ) между таблицами учебной БД. Произвести проверку правильности выполнения каждого задания.

10. Составить отчет. В отчете представить тексты запросов, построенных при выполнении общего и индивидуального заданий, а также результаты их выполнения и проверки правильности.

11. Защитить лабораторную работу, построив и результативно выполнив под контролем информационной системы [32] заданное количество запросов к учебной БД.

Варианты индивидуальных заданий

Вариант 1

1. Увеличить на 50 руб. все значения начислений для абонента с ФИО Мищенко Е. В.

2. Удалить все факты оплаты с суммой менее 25 руб.
3. Изменить номер телефона абонента с номером лицевого счета '015527' на любой другой.
4. Удалить информацию обо всех начислениях по услуге Газоснабжение за декабрь 2015 г.
5. Для абонентов с минимальным значением оплаты уменьшить значения начислений на 10 %.

Вариант 2

1. Увеличить на 20 % значения всех начислений за 2018 г. для абонентов, у которых значения начислений за февраль 2017 г. меньше средних значений их оплат за период 2015—2017 гг. (все годы учитываются полностью) по услуге Газоснабжение.
2. Удалить информацию обо всех начислениях для абонентов, проживающих на улице МОСКОВСКОЕ ШОССЕ.
3. Удалить информацию обо всех платежах абонентов, чье среднее значение оплат меньше 150 руб.
4. Перевести всех абонентов, проживающих на улице с названием МОСКОВСКАЯ УЛИЦА, на улицу с названием НОВАЯ УЛИЦА.
5. Удалить исполнителей, у которых есть хотя бы одна непогашенная заявка.

Вариант 3

1. Перевести факты оплаты, сделанные с 05.02.2017 по 26.11.2017, на текущую дату.
2. Удалить информацию о платежах всех абонентов, в ФИО которых встречается буква К (без учета регистра символов).
3. Поменять ФИО исполнителя ремонтных заявок Шубин В. Г. на свои ФИО.
4. Удалить все ремонтные заявки, которые были поданы абонентом с ФИО Конюхов В. С.
5. Уменьшить на 5 руб. значение начислений за 2018 г. для абонентов, которые в 2015 г. платили два и более раз.

Вариант 4

1. Увеличить на 50 % все значения начислений за 2018 г.
2. Удалить все факты оплаты со значением оплаты между 5 и 115 руб.
3. Для всех ремонтных заявок установить дату выполнения 11.01.2018 и поставить отметку о погашении.
4. Удалить информацию обо всех неисправностях из справочника неисправностей газового оборудования.
5. Удалить все ремонтные заявки, в которых в качестве неисправности указано «Не горит АГВ».

Вариант 5

1. Изменить дату подачи всех ремонтных заявок абонента с ФИО Свирина З. А. на 01.01.2017.

2. Удалить информацию о платежах всех абонентов, проживающих на улице ВОЙКОВ ПЕРЕУЛОК и имеющих три и более факта оплаты за 2015—2016 гг.

3. Для всех абонентов, проживающих по КУТУЗОВА УЛИЦА, удалить значения номеров телефонов.

4. Удалить все заявки из таблицы ремонтных заявок.

5. Для абонентов, чьи ремонтные заявки обслуживает Шубин В. Г., удалить все начисления за услугу «Газоснабжение».

Вариант 6

1. Назначить исполнителя с ФИО Стародубцев Е. М. ответственным за обслуживание всех ремонтных заявок, поданных абонентом Мищенко Е. В.

2. Удалить все факты оплаты со значениями меньше 100 руб.

3. Изменить номер дома, в котором проживает абонент с ФИО Маркова В. П., прибавив к старому номеру число 5.

4. Удалить всю информацию о начислениях за декабрь 2015 и 2016 гг.

5. Удалить информацию обо всех начислениях тех абонентов, которыми сделано более трех оплат со значениями ниже среднего значения оплат всех абонентов.

Вариант 7

1. Увеличить на 10 руб. все значения начислений за 2017 г. всем абонентам.

2. Удалить всех исполнителей, за которыми закреплено менее четырех ремонтных заявок.

3. Перевести всех абонентов с улиц с названием МОСКОВСКАЯ УЛИЦА и МОСКОВСКОЕ ШОССЕ на улицу с названием ВОЙКОВ ПЕРЕУЛОК.

4. Удалить информацию о платежах всех абонентов с номером лицевого счета менее 115705.

5. Удалить информацию о платежах всех абонентов, у которых нет невыполненных ремонтных заявок.

Вариант 8

1. Перевести все начисления за месяцы позже июля 2016 г. на январь 2017 г.

2. Удалить всех исполнителей, в ФИО которых встречается буква А (без учета регистра символов).

3. Изменить ФИО абонента с Маркова В. П. на Журавлева В. П.

4. Удалить все начисления за 2016 г. для абонентов, проживающих по ТАТАРСКАЯ УЛИЦА.

5. Увеличить на 20 % значения начислений для абонентов, которые подали более четырех ремонтных заявок.

Вариант 9

1. Увеличить на три номера квартир всех абонентов.

2. Удалить все факты оплаты со значениями между 50 руб. и средним значением оплат.

3. Удалить все факты оплаты для абонента с ФИО Тулупова М. И.
4. Установить для всех ремонтных заявок дату выполнения равной 12.01.2017.
5. Удалить информацию о начислениях всех абонентов, которые в 2015 г. платили два раза.

Вариант 10

1. Ко всем номерам телефонов абонентов добавить код города Рязань в формате: 4912-номер телефона абонента.
2. Удалить ремонтные заявки абонентов, проживающих на улице ВОЙКОВ ПЕРЕУЛОК.
3. Для ремонтных заявок, поданных в 2015 г., установить в качестве неисправности «Неизвестна».
4. Удалить все улицы из справочника улиц.
5. Для всех ремонтных заявок с неисправностью «Неисправен газовый счетчик» установить в качестве ответственного исполнителя с ФИО Школьников С. М.

Лабораторная работа № 6.

Программирование хранимых процедур

Цель работы. Приобретение практических навыков программирования ХП.

Лабораторное задание. Изучить теоретическую часть, изложенную в параграфах 6.1 и 6.2. Разработать и выполнить хранимые процедуры согласно заданиям. Защитить лабораторную работу.

Порядок выполнения работы

1. Подключиться к учебной БД.
2. Выполнить SQL-сценарий, приведенный в подпараграфе 6.2.4, добавляющий в учебную БД таблицу Ftable, хранимые процедуры Factorial и FactorialSet.
3. Изменить (используя запрос ALTER PROCEDURE) процедуру FactorialSet таким образом, чтобы при ее вызове не требовалось указывать входные параметры, а вычисление факториала чисел от 1 до 31 и занесение вычисленных значений в таблицу Ftable происходили автоматически. При этом получение очередного числа для вычисления его факториала должно осуществляться с помощью генератора. Перед каждым запуском на выполнение процедуры FactorialSet необходимо устанавливать начальное значение генератора равным нулю.
4. Разработать селективную и выполняемую процедуры в соответствии с вариантом индивидуального задания. Обращать возможные некорректные или отсутствующие значения, выдавая соответствующее сообщение об исключительной ситуации. Проверить работу созданных ХП.

5. Составить отчет. В отчете представить тексты ХП, созданных при выполнении общего и индивидуального заданий, результаты их выполнения и проверки правильности при различных исходных данных, в том числе ошибочных и (или) отсутствующих.

6. Защитить лабораторную работу, положительно ответив под контролем информационной системы [32] на заданное количество тестовых вопросов по теме настоящей лабораторной работы.

Варианты индивидуальных заданий

Вариант 1

1. Создать селективную процедуру, которая возвращает для всех платежей код платежа, дату, значение и ФИО абонента, совершившего платеж. Выполнить процедуру. Изменить определение процедуры, добавив в качестве входного параметра номер лицевого счета абонента. Выполнить процедуру.

2. Создать выполняемую процедуру, которая для заданного месяца и года производит абонентам по услуге «Газоснабжение» начисление за выполненные им в этом месяце ремонтные заявки. Суммы начислений принять равными 100 руб. для неисправностей, связанных с АГВ или печной горелкой, 75 руб. — с водогрейной колонкой, 50 руб. — с газовой плитой и 40 руб. — со счетчиком.

3. Разработать скрипт, позволяющий модифицировать структуру таблицы Request, заменяя столбцы с датой регистрации и выполнения заявки на один столбец.

Вариант 2

1. Создать селективную процедуру, которая возвращает всю информацию о платежах. Выполнить процедуру. Изменить определение процедуры, добавив в качестве входных параметров год и месяц, за которые произведены платежи. Выполнить процедуру.

2. Создать выполняемую ХП, которая для заданного абонента изменяет значения начислений за нужные месяц и год на максимальное значение начисления для услуги из изменяемой строки. Входные параметры: ФИО абонента, месяц и год начисления.

3. Разработать скрипт, позволяющий модифицировать структуру таблицы Request, заменяя столбец с датой регистрации заявки на три столбца.

Вариант 3

1. Создать селективную процедуру, возвращающую ФИО каждого абонента и значения его начислений, сопровождая вывод информацией о телефоне абонента и месяце, за который начислено. Выполнить процедуру. Изменить определение процедуры, добавив в качестве входных параметров номер лицевого счета абонента и год начисления.

2. Добавить в справочник услуг услугу с наименованием Пени. Создать выполняемую процедуру, которая анализирует число ме-

сяцев задолженности у каждого абонента по услуге, задаваемой параметром, по следующему алгоритму. Если у абонента за определенный месяц есть начисление, но нет оплаты, то это означает, что у него имеется 1 месяц задолженности. Выполнить начисление (в процедуре) каждому абоненту по услуге Пени, если реальное число месяцев задолженности превышает количество месяцев задолженности, задаваемое входным параметром. Размер пени рассчитывать, исходя из действующего в настоящее время процента ставки рефинансирования Центрального Банка РФ от суммы задолженности. Для этого использовать следующую формулу:

$$\text{ПЕНИ} = \text{Сумма_задолженности} * \text{Количество_дней_просрочки} / 300 * \text{Ставка_рефинансирования}$$

Вывести результаты расчета.

Вариант 4

1. Создать селективную процедуру, возвращающую значения и даты платежей, сделанных абонентами, Выполнить процедуру. Изменить определение процедуры, добавив в качестве входного параметра ФИО абонента. Выполнить процедуру.

2. Создать процедуру для анализа возможной связи неисправности газового оборудования и улиц. Входным параметром является число заявок и тип преимущественной неисправности. Результатом должен быть список улиц, у абонентов которых имеется количество заявок с заданной неисправностью, превышающее заданное число.

3. Создать процедуру, возвращающую для каждого абонента итоговое сальдо по всем услугам (разница между суммами значений начислений и оплат).

Вариант 5

1. Создать селективную процедуру, возвращающую ФИО абонентов, которые имеют значения платежей выше среднего по каждой услуге. Выполнить процедуру. Изменить определение процедуры, добавив в качестве входного параметра наименование услуги. Выполнить процедуру.

2. Создать таблицу, позволяющую хранить в одной строке следующую информацию: лицевой счет, ФИО, полная «красивая» строка адресных данных, текущий баланс лицевого счета, дата последней оплаты, общее число выполненных ремонтных заявок. Создать выполняемую процедуру по заполнению добавленной таблицы.

3. Разработать скрипт, позволяющий модифицировать структуру таблицы Request заменяя столбец с датой выполнения заявки на три столбца.

Вариант 6

1. Создать селективную процедуру, которая возвращает сумму начислений и ФИО абонентов, полностью оплативших услуги.

Выполнить процедуру. Изменить определение процедуры, добавив в качестве входного параметра год, за который произведены начисления и платежи. Выполнить процедуру.

2. Добавить в справочник услуг услугу с наименованием *Списание задолженности*. Создать процедуру, которая, анализируя входной параметр — сумма, будет помещать для каждого абонента в таблицу оплат *Paysumma* записи со значением оплаты, равным размеру имеющейся задолженности по соответствующей услуге, но не более введенного значения, а в качестве даты оплаты, месяца и года — текущие дату, месяц и год соответственно. После выполнения процедура должна возвращать число обработанных абонентов и общую сумму списанной задолженности.

Вариант 7

1. Создать таблицу, позволяющую хранить следующую информацию: код исполнителя ремонтной заявки (первичный ключ таблицы), ФИО исполнителя и количество выполненных им заявок. Создать процедуру, заполняющую созданную таблицу данными о заявках тех абонентов, которые оплачивали за заданную услугу 2 и более раз.

2. Создать две хранимых процедуры. Первая процедура по заданному номеру лицевого счета абонента должна возвращать код исполнителя, который последним выполнял ремонтную заявку, поданную данным абонентом. Вторая процедура должна заполнять созданную таблицу, при этом используя результаты вызова первой процедуры и перезаписывая информацию о количестве заявок исполнителя, если по этому исполнителю уже есть данные в таблице.

Вариант 8

1. Создать селективную процедуру, которая возвращает дату приема и выполнения заявок. Выполнить процедуру. Изменить определение процедуры, добавив в качестве входного параметра ФИО мастера. Выполнить процедуру.

2. Создать таблицу, позволяющую хранить следующую информацию: номер лицевого счета абонента, ФИО абонента, дата первого начисления (последний день месяца, за который было выполнено первое начисление абоненту) и наименование соответствующей услуги. Создать выполняемую процедуру, заполняющую созданную таблицу данными для всех абонентов, у которых имеются начисления по услуге, наименование которой задается входным параметром.

3. Разработать скрипт, позволяющий модифицировать структуру таблицы *Abonent*, заменяя 3 столбца с адресом на один столбец.

Вариант 9

1. Создать селективную процедуру, которая возвращает всю информацию об абонентах, имеющих ремонтные заявки. Выполнить процедуру. Изменить определение процедуры, добавив в качестве

входного параметра дату регистрации заявок. Выполнить процедуру.

2. Создать таблицу, позволяющую хранить следующую информацию: номер лицевого счета абонента, ФИО абонента, дата первой оплаты. Создать выполняемую процедуру, заполняющую данную таблицу данными по всем абонентам, у которых есть оплаты.

3. Разработать скрипт, позволяющий модифицировать структуру таблицы Abonent, заменяя столбцы с адресом и номером телефона на один столбец.

Вариант 10

1. Создать таблицу, позволяющую хранить следующую информацию: код исполнителя ремонтной заявки (первичный ключ таблицы), ФИО исполнителя и количество выполненных им заявок. Создать процедуру, заполняющую созданную таблицу данными о заявках тех абонентов, у которых сумма значений плат за предоставляемую услугу более заданной.

2. Создать 2 хранимые процедуры. Первая процедура по заданному номеру лицевого счета абонента должна возвращать код исполнителя, который первым выполнял ремонтную заявку, поданную данным абонентом. Вторая процедура должна заполнять созданную таблицу, при этом используя результаты вызова первой процедуры и перезаписывая информацию о количестве заявок исполнителя, если по этому исполнителю уже есть данные в таблице.

Вариант 11

1. Создать хранимую процедуру, которая выводит следующую информацию: ФИО абонента, название улицы, номер дома, номер квартиры, тип возникших неисправностей газового оборудования. Выполнить процедуру. Изменить определение процедуры, добавив в качестве входного параметра наименование неисправности. Выполнить процедуру.

2. Создать процедуру, которая выводит разницу между общей суммой начислений и общей суммой плат каждого абонента, а также его ФИО, улицу, номер дома, номер квартиры, телефон.

3. Разработать скрипт, позволяющий модифицировать структуру таблицы Request, объединяя столбцы с датой регистрации и датой выполнения заявки в один столбец.

Вариант 12

1. Создать селективную процедуру, возвращающую название улиц. Выполнить процедуру. Изменить определение процедуры, добавив в качестве входного параметра число проживающих абонентов. Выполнить процедуру.

2. Создать процедуру, выполняющую уменьшение на 15 % значений плат за определенную услугу и год (задаются параметрами) у абонентов, имеющих отрицательный результирующий баланс по этой услуге.

Вариант 13

1. Создать селективную процедуру, обеспечивающую вывод в одну строку через запятую всех значений столбца таблицы. Выполнить процедуру. Изменить определение процедуры, добавив в качестве входных параметров имена таблицы и столбца. Выполнить процедуру.

2. Создать процедуру, выполняющую возведение в целую степень целого числа с выводом результатов, начиная с нулевой степени с шагом 1. Исходное число и значение степени задаются параметрами.

3. Разработать скрипт, позволяющий модифицировать структуру таблицы PaySumma, заменяя столбцы с месяцем и годом периода оплаты одним столбцом с первым числом месяца и года этого периода.

Вариант 14

1. Создать селективную процедуру, возвращающую все данные из заданной таблицы. Выполнить процедуру. Изменить определение процедуры, добавив в качестве входного параметра имя выводимого столбца. Выполнить процедуру.

2. Создать процедуру, выполняющую увеличение на 20 % значений начислений за определенную услугу и год (задаются параметрами) у абонентов, имеющих отрицательный результирующий баланс (разница между суммами начислений и оплат) по этой услуге.

3. Разработать скрипт, позволяющий модифицировать структуру таблицы NachislSumma, заменяя столбцы с месяцем и годом периода начисления одним столбцом с первым числом месяца и года этого периода.

Вариант 15

1. Разработать скрипт для добавления двух таблиц Remont и Rab_Remont. Таблица Remont содержит перечень типовых ремонтных работ, выполняемых при устранении неисправностей газового оборудования (код работы, наименование работы). Заполнить таблицу Remont данными, выбранными самостоятельно. Таблица Rab_Remont является списком ремонтных работ, выполненных по каждой ремонтной заявке (код, код заявки, код ремонтной работы). Таблицы Remont и Request должны являться родительскими по отношению к таблице Rab_Remont.

2. Создать процедуру ДОБАВИТЬ_ЗАЯВКУ, имеющую входные параметры, соответствующие столбцам таблицы Request. Процедура должна добавлять переданную в параметрах заявку в таблицу Request и проводить полную диагностику ошибок (правильность внешних ключей, правильность ограничений, корректность добавляемых значений).

3. Создать процедуру ДОБАВИТЬ_РЕМОНТНЫЕ_РАБОТЫ, имеющую входные параметры, соответствующие столбцам таблицы Rab_

Remont. Процедура должна добавлять переданную в параметрах ремонтную работу в таблицу Rab_Remont и проводить полную диагностику ошибок (правильность внешних ключей, правильность ограничений, корректность добавляемых значений).

Лабораторная работа № 7. Программирование триггеров

Цель работы. Приобретение практических навыков программирования триггеров.

Лабораторное задание. Изучить теоретическую часть, изложенную в параграфе 6.6. Разработать триггеры согласно варианту задания и проверить их работоспособность. Защитить лабораторную работу.

Порядок выполнения работы

1. Подключиться к учебной БД.
2. Определить способ реализации ограничений ссылочной целостности между таблицами учебной БД (декларативный или процедурный), заданными вариантом индивидуального задания в лабораторной работе № 1.
3. Выполнить индивидуальное задание согласно варианту.
 - 3.1. Разработать и создать триггер DML для формирования значений первичных ключей при добавлении новых записей в таблицу учебной БД (что должно найти свое отражение в названии триггера). При этом получение очередного значения первичного ключа должно автоматически осуществляться с помощью генератора как следующее по порядку значение первичного ключа таблицы. Извлечь и изучить информацию об использованном генераторе.
 - 3.2. Разработать и создать триггер DML для обеспечения целостности данных или реализации бизнес-логики.
 - 3.3. Разработать и создать триггер, отвечающий за реализацию ссылочной целостности между таблицами учебной БД.
4. Разработать и создать триггер DML, реализующий бизнес-правило, сформулированное самостоятельно.

Примечание. Триггеры в пп. 3.2, 3.3 и 4 должны быть разработаны для ограничений целостности, которые не могут быть реализованы декларативно, т. е. с помощью ограничений CHECK, DEFAULT, FOREIGN KEY REFERENCES, NOT NULL, PRIMARY KEY и UNIQUE.

5. Разработать и создать триггер DDL, реализующий самостоятельно сформулированное задание.

6. Разработать и создать триггер БД, реализующий самостоятельно сформулированное задание.

7. Проверить работоспособность созданных триггеров путем внесения соответствующих изменений БД.

8. Составить отчет, в котором представить тексты созданных триггеров и результаты их работы.

9. Защитить лабораторную работу, положительно ответив под контролем информационной системы [32] на заданное число тестовых вопросов по теме настоящей лабораторной работы.

Варианты индивидуальных заданий

Вариант 1

1. Services.

2. Запрет удаления исполнителей ремонтных заявок, имеющих хотя бы одну непогашенную заявку.

3. Реализация правила ссылочной целостности типа U:R между таблицами Abonent и Request.

Вариант 2

1. Disrepair.

2. При обновлении фамилии абонента фиксировать в некоторой таблице сведения о его старой и новой фамилиях, а также дату и время выполнения этих изменений и имена пользователей, которые произвели обновления в таблице.

3. Реализация правила ссылочной целостности типа D:R между таблицами Street и Abonent.

Вариант 3

1. Street.

2. Запрет изменения адреса абонента, имеющего задолженность по оплате услуг.

3. Изменить существующий триггер, отвечающий за реализацию правила ссылочной целостности типа D:R между таблицами Services и PaySumma, на правило D:SN.

Вариант 4

1. Executor.

2. Запрет регистрации ремонтной заявки от абонента, допустившего просрочку по платежам (оплачивает позже 10 числа месяца следующего за оплачиваемым) более трех раз.

3. Реализация правила ссылочной целостности типа U:R между таблицами Services и NachislSumma.

Вариант 5

1. Disrepair.

2. Запрет регистрации заявки с неисправностью «Засорилась водогрейная колонка» от абонентов, проживающих на улице ВОЙКОВ ПЕРЕУЛОК (код улицы неизвестен).

3. Реализация правила ссылочной целостности типа D:R между таблицами Disrepair и Request.

Вариант 6

1. Street.

2. Разработать триггер, который после удаления неисправности из таблицы Disrepair удаляет ремонтные заявки с соответствующей неисправностью из таблицы Request.

3. Реализация правила ссылочной целостности типа D:R между таблицами Abonent и Request.

Вариант 7

1. Disrepair.

2. При удалении исполнителя ремонтных заявок фиксировать изменения в некоторой таблице. Таблица должна содержать номер удаленного исполнителя, ФИО абонентов, которых он обслуживал, дату и время удаления.

3. Реализация правила ссылочной целостности типа U:R между таблицами Street и Abonent.

Вариант 8

1. Services.

2. Фиксация в некоторой таблице имен пользователей, которые обновляют данные в таблице PaySumma, и даты выполнения изменения.

3. Реализация правила ссылочной целостности типа U:R между таблицами Executor и Request.

Вариант 9

1. Disrepair.

2. Запрет изменения адреса абонента, имеющего задолженность по оплате услуги «Газоснабжение».

3. Реализация правила ссылочной целостности типа U:R между таблицами Abonent и PaySumma.

Вариант 10

1. Street.

2. При изменении данных в таблице NachislSumma фиксировать в некоторой таблице дату и время изменения, имя пользователя, осуществившего эти изменения, а также новые и старые значения строк, подвергшихся изменению.

3. Реализация правила ссылочной целостности типа D:R между таблицами Executor и Request.

Вариант 11

1. Executor.

2. Запрет вставки строки в таблицу Abonent, если существует исполнитель ремонтных заявок, фамилия которого начинается на ту же букву, что и фамилия вставляемого абонента.

3. Реализация правила ссылочной целостности D:C между таблицами Street и Abonent.

Вариант 12

1. Disrepair.

2. Запрет вставки строки в таблицу Street, если существует улица с таким же названием.

3. Изменить существующий триггер, отвечающий за реализацию правила ссылочной целостности типа D:R между таблицами Abonent и PaySumm, на правило D:SN.

Вариант 13

1. Executor.

2. Запрет вставки строки в таблицу Request, если дата выполнения заявки раньше даты ее регистрации.

3. Реализация правила ссылочной целостности типа D:C между таблицей Services и таблицами NachislSumma и PaySumma.

Вариант 14

1. Street.

2. Запрет регистрации ремонтной заявки в субботу и воскресенье.

3. Создать триггер, который рассчитывает среднюю задолженность по каждой услуге и выдает сообщение при превышении заданного порога отклонения вводимого значения начисления или оплаты в зависимости от среднего значения задолженности. При этом происходит заполнение некоторой таблицы.

Вариант 15

1. Executor.

2. Запрет регистрации оплаты в воскресенье.

3. Создать триггер, который рассчитывает среднее значение оплаты в заданный день и выдает сообщение при превышении заданного порога отклонения вводимого значения оплаты в зависимости от среднего значения. При этом происходит заполнение некоторой таблицы.

Лабораторная работа № 8. Программирование запросов управления доступом

Цель работы. Приобретение практических навыков построения запросов управления доступом к объектам БД.

Лабораторное задание. Изучить теоретическую часть, изложенную в гл. 7. Выполнить общее и индивидуальное задания. Защитить лабораторную работу.

Порядок выполнения работы

1. Подключиться к учебной БД как системный администратор. Зарегистрировать на сервере двух новых пользователей с выбранными самостоятельно именами и паролями.

2. Подключиться к учебной БД как системный администратор. Создать три роли с именами Delete_Role, Insert_Role, Select_Role. Присвоить первой роли привилегию DELETE на три произвольные таблицы учебной БД, второй роли — привилегию INSERT и третьей

роли — привилегию SELECT. Передать право на использование роли Select_Role одному из зарегистрированных пользователей. Подключиться к БД под его именем с указанием имени той роли, правом на использование которой он обладает. Проверить у этого пользователя наличие привилегий, которые он приобрел при получении права использовать конкретную роль. Записать сообщение об ошибке, которая возникает при попытке пользователя осуществить доступ к таблице в случае отсутствия прав доступа. Отсоединиться от БД.

3. Подключиться к БД как системный администратор. Извлечь информацию о предоставленных привилегиях с помощью соответствующих средств используемого клиента и зафиксировать ее в отчете.

4. Создать процедуру выбора, которая должна в выходных параметрах возвращать ФИО конкретного абонента, дату и сумму оплаты, месяц и год, за который произведена оплата. Передать другому пользователю (у которого еще нет никаких привилегий) право на выполнение этой ХП. При этом пользователь должен иметь привилегию только на выполнение данной процедуры, не имея никаких других привилегий.

5. Извлечь информацию о предоставленных привилегиях и зафиксировать ее в отчете.

6. Выполнить отмену всех предоставленных выше привилегий доступа.

7. Передать одному из зарегистрированных пользователей привилегию SELECT на представления, созданные в лабораторной работе № 4, привилегию EXECUTE на процедуры, созданные в лабораторной работе № 6. Передать триггерам, созданным в лабораторной работе № 7, привилегию UPDATE на одну из таблиц учебной БД.

8. Извлечь информацию о предоставленных привилегиях и зафиксировать ее в отчете.

9. Выполнить отмену всех предоставленных выше привилегий доступа.

10. Выполнить индивидуальное задание.

11. Составить отчет. В отчете представить тексты всех SQL-запросов, использованных при выполнении лабораторной работы, текст скрипта на создание процедуры выбора в п. 4, а также информацию о привилегиях пользователей, предоставленных им на всех этапах выполнения лабораторной работы.

12. Защитить лабораторную работу, положительно ответив под контролем информационной системы [32] на заданное количество тестовых вопросов по теме настоящей лабораторной работы.

Варианты индивидуальных заданий

Во всех вариантах создать роль в соответствии с заданием. Передать право на ее использование одному из предварительно зарегистрированных пользователей. Подключиться к БД под именем

зарегистрированного пользователя с указанием имени той роли, правом на использование которой он обладает. Проверить наличие у него привилегий, которые он приобрел при получении права использовать конкретную роль. Записать сообщение об ошибке, которая возникает при попытке пользователя осуществить доступ к таблице в случае отсутствия прав доступа.

Вариант 1

Создать роль Manager. Роль должна обладать правами полного доступа ко всем таблицам БД.

Вариант 2

Создать роль Visitor. Роль должна обладать привилегией SELECT на таблицы Street, Services, Disrepair и Abonent.

Вариант 3

Создать роль Dispatcher. Роль должна обладать привилегией SELECT на все таблицы БД и привилегией INSERT на таблицу Request.

Вариант 4

Создать роль Chief. Роль должна обладать привилегией SELECT на все таблицы БД и привилегией UPDATE на таблицы Executor, NachislSumma, Disrepair.

Вариант 5

Создать роль Cashier. Роль должна обладать привилегией SELECT на все таблицы БД, кроме Executor, Disrepair и Request, и привилегией INSERT на таблицу PaySumma.

Вариант 6

Создать роль Boss. Роль должна обладать привилегией SELECT на все таблицы БД и привилегией INSERT на таблицы БД, кроме Executor, Disrepair и Request.

Вариант 7

Создать роль Executor и обеспечить ее всеми необходимыми привилегиями для выполнения одной из имеющихся ХП.

Вариант 8

Создать роль PaysControl и обеспечить ее всеми необходимыми привилегиями для выполнения одной из имеющихся ХП.

Вариант 9

Создать роль InserterAbonent и обеспечить ее всеми необходимыми привилегиями для работы с таблицей Abonent. При этом не давать роли привилегии на чтение никаких других таблиц.

Вариант 10

Создать роль StreetDeleter и обеспечить ее всеми необходимыми привилегиями для удаления строк из таблицы Street. При этом не давать роли привилегии на чтение никаких других таблиц.

Вариант 11

Создать роль Control и обеспечить ее всеми необходимыми привилегиями для выполнения одного из имеющихся триггеров.

Приложение А.

Описание учебной базы данных

Жилищно-коммунальное хозяйство является одной из основных составляющих (компонентов, элементов, областей) народного хозяйства РФ, охватывающей многоотраслевой производственно-технический комплекс, потребность в продукции которого практически не ограничена. Индустрия ЖКХ представлена жилыми, общественными зданиями, эксплуатационными, управляющими, ремонтно-строительными, транспортными, ресурсоснабжающими и другими предприятиями, составляющими сложную социально-экономическую систему, от результативности функционирования которой зависит развитие городских объектов и состояние среды обитания жителей городов и других населенных пунктов.

Расчет и прием жилищно-коммунальных платежей при кажущейся элементарности сервиса — важная составляющая повседневного быта граждан. Каждый из них для удовлетворения своих жизненных потребностей потребляет или расходует коммунальные ресурсы (вода, газ, тепло, электроэнергия) и является абонентом одной из управляющих компаний. В случае возникновения неисправности источников коммунальных ресурсов (поломка газового, отопительного, сантехнического или электрооборудования) он отправляет аварийную или ремонтную заявку по устранению возникшей неисправности.

Компании, оказывающие такие услуги и поставляющие коммунальные ресурсы, ежемесячно производят начисление платы соответствующего размера за них, а обязанность абонентов — своевременно и регулярно их оплачивать.

Правила предоставления жилищно-коммунальных услуг собственникам и пользователям помещений в многоквартирных и жилых домах, а также порядок расчета и перерасчета размера платы за коммунальные ресурсы с использованием приборов учета и при их отсутствии регулируются соответствующими постановлениями правительства.

Следует отметить существенное усложнение в последнее время алгоритмов расчета размера платы за коммунальные ресурсы и жилищно-коммунальные услуги. Прежде всего, это:

- расчет размера платы за коммунальную услугу при отсутствии показаний прибора учета (счетчика), исходя из рассчитанного сред-

немесячного или нормативного объема расхода потребителем коммунального ресурса (объема потребления коммунальной услуги);

- использование в расчетах дифференцированной ставки процента пени в зависимости от числа дней образования просроченной задолженности;

- применение дифференцированного повышающего коэффициента для абонентов, имеющих техническую возможность, но не установивших приборы учета коммунальных ресурсов;

- расчет объемов коммунальных ресурсов, расходуемых на общедомовые нужды;

- расчет с применением многотарифного учета объемов расходуемого коммунального ресурса;

- дифференцированный расчет за централизованное отопление в зависимости от года и материала постройки дома, а также его этажности;

- обеспечение интеграции информационных систем муниципального и регионального уровней с государственной информационной системой жилищно-коммунального хозяйства (ГИС ЖКХ).

Учебная БД представляет собой очень упрощенный пример информационной модели расчетно-аналитической составляющей «Абонент» расчетно-платежного комплекса «Абонент+», которая используется для информационного обеспечения деятельности организаций по оказанию населению жилищно-коммунальных услуг и расчетам за них [35]. Учебная БД состоит из восьми таблиц: пяти таблиц-справочников и трех информационных таблиц (таблиц фактов).

В учебной БД используются такие таблицы-справочники:

1) **Street**. Справочник улиц города, в домах которых проживают абоненты. Столбцы таблицы:

- StreetCD — уникальный код улицы (первичный ключ),
- StreetNM — название улицы, расшифровывающее код улицы;

2) **Abonent**. Справочник абонентов города. Столбцы таблицы:

- AccountCD — номер лицевого счета абонента, уникальным образом идентифицирующий каждого из абонентов (первичный ключ),

- StreetCD — код улицы, на которой проживает абонент (внешний ключ, ссылающийся на первичный ключ таблицы Street),

- HouseNo — номер дома, в котором проживает абонент,

- FlatNo — номер квартиры,

- Fio — фамилия, имя и отчество абонента в формате «Фамилия И. О.»,

- Phone — номер телефона;

3) **Services**. Справочник услуг, оказываемых абонентам жилищно-коммунальными организациями. Столбцы таблицы:

- ServiceCD — код услуги (первичный ключ),

- ServiceNM — наименование услуги;

4) **Disrepair.** Справочник типовых неисправностей газового оборудования абонентов. Столбцы таблицы:

- FailureCD — код неисправности газового оборудования (первичный ключ),
- FailureNM — наименование неисправности газового оборудования;

5) **Executor.** Справочник исполнителей заявок, поданных абонентами на ремонт газового оборудования. Исполнителями являются работники ремонтной службы газораспределительной организации, оказывающей соответствующие услуги. Столбцы таблицы:

- ExecutorCD — уникальный код, идентифицирующий исполнителей ремонтных заявок (первичный ключ),
- Fio — фамилия, имя и отчество исполнителя в формате «Фамилия И. О.».

В качестве информационных таблиц учебной БД выделены следующие:

1) **NachislSumma.** Таблица для хранения информации о размерах ежемесячных начислений плат абонентам за оказанные им услуги (которые расшифровываются в справочнике услуг Services). Плата за ремонт газового оборудования производится по факту оказания услуги, и начисление за него не производится. Столбцы таблицы:

- NachislFactCD — уникальный идентификатор факта начисления (первичный ключ),
- AccountCD — номер лицевого счета абонента, которому было сделано начисление (внешний ключ, ссылающийся на первичный ключ таблицы Abonent),
- ServiceCD — код услуги, за которую выполнено начисление (внешний ключ, ссылающийся на первичный ключ таблицы Services),
- NachislSum — значение начисленной суммы,
- NachislMonth — номер месяца, за который произведено начисление,
- NachislYear — год, за месяц которого выполнено начисление;

2) **PaySumma.** Таблица для хранения значений оплат, внесенных абонентами за оказанные им услуги. Для каждого факта оплаты по какой-либо услуге указывается дата осуществления оплаты, оплачиваемые месяц и год. Столбцы таблицы:

- PayFactCD — уникальный идентификатор факта оплаты по услуге Request (первичный ключ),
- AccountCD — номер лицевого счета абонента, оплатившего оказанную ему услугу (внешний ключ, ссылающийся на первичный ключ таблицы Abonent),
- ServiceCD — код оплаченной услуги (внешний ключ, ссылающийся на первичный ключ таблицы Services),

- PaySum — значение оплаченной суммы,
- PayDate — дата оплаты,
- PayMonth — номер оплачиваемого месяца,
- PayYear — оплачиваемый год.

Таким образом, при сопоставлении этой информации по конкретному абоненту с данными, хранящимися в таблице NachislSumma, можно вычислить размер долга или переплаты у данного абонента на указанный месяц года;

3) **Request.** Таблица для хранения информации о заявках абонентов на ремонт газового оборудования. Каждая ремонтная заявка характеризуется номером лицевого счета абонента (расшифровка в справочнике Abonent), заявившего определенную неисправность газового оборудования (расшифровка в справочнике Disrepair) в его доме или квартире, исполнителем ремонтной работы (справочник Executor), датой регистрации заявки, датой выполнения ремонта и признаком погашения (закрытия). Столбцы таблицы:

- RequestCD — уникальный код ремонтной заявки (первичный ключ),
- AccountCD — номер лицевого счета абонента, подавшего данную ремонтную заявку (внешний ключ, ссылающийся на первичный ключ таблицы Abonent),
- FailureCD — код неисправности газового оборудования, заявленной абонентом в данной ремонтной заявке (внешний ключ, ссылающийся на первичный ключ таблицы Disrepair),
- ExecutorCD — код исполнителя, ответственного за выполнение данной ремонтной заявки (внешний ключ, ссылающийся на первичный ключ таблицы Executor),
- IncomingDate — дата поступления заявки,
- ExecutionDate — дата выполнения заявки,
- Executed — столбец логического типа, признак погашения заявки.

В общем случае зарегистрированная ремонтная заявка может быть:

- не назначена ни одному из исполнителей, а следовательно, не выполнена и не погашена;
- назначена одному из исполнителей, но не выполнена им, а следовательно, не погашена;
- назначена одному из исполнителей, выполнена им, но не погашена;
- назначена одному из исполнителей, выполнена им и погашена.

ER-модели учебной БД на логическом и физическом уровнях представлены соответственно на рис. 1.5 и 1.6 (см. гл. 1, ч. 1).

Для обработки исключительных ситуаций созданы исключения: Del_ Restrict, Ins_ Restrict и Upd_ Restrict.

Ниже приводятся таблицы учебной БД. Весь материал учебника, все запросы основаны именно на этих данных. Анализ таблиц поможет понять логику получения результатов.

Объектам учебной БД, согласно стилю программирования, даны вполне осмысленные имена. Это, в свою очередь, привело к необходимости использования довольно длинных имен, что не является недостатком при их задании, так как современные клиентские утилиты производят контекстный поиск при вводе по первым символам имени объекта. Таблицы фактов учебной БД имеют большое число строк. Это сделано для того, чтобы приблизить ее к реальности.

Таблица А.1

Данные таблицы Street

StreetCD	StreetNM
1	ЦИОЛКОВСКОГО УЛИЦА
2	НОВАЯ УЛИЦА
3	ВОЙКОВ ПЕРЕУЛОК
4	ТАТАРСКАЯ УЛИЦА
5	ГАГАРИНА УЛИЦА
6	МОСКОВСКАЯ УЛИЦА
7	КУТУЗОВА УЛИЦА
8	МОСКОВСКОЕ ШОССЕ

Таблица А.2

Данные таблицы Abonent

AccountCD	StreetCD	HouseNo	FlatNo	Fio	Phone
005488	3	4	1	Аксенов С. А.	556893
015527	3	1	65	Конюхов В. С.	761699
080047	8	39	36	Шубина Т. П.	257842
080270	6	35	6	Тимошкина Н. Г.	321002
080613	8	35	11	Лукашина Р. М.	254417
115705	3	1	82	Мищенко Е. В.	769975
126112	4	7	11	Маркова В. П.	683301
136159	7	39	1	Свирина З. А.	NULL
136160	4	9	15	Шмаков С. В.	NULL
136169	4	7	13	Денисова Е. К.	680305
443069	4	51	55	Стародубцев Е. В.	683014
443690	7	5	1	Тулупова М. И.	214833

Таблица А.3

Данные таблицы Services

ServiceCD	ServiceNM
1	Газоснабжение
2	Электроснабжение
3	Теплоснабжение
4	Водоснабжение

Таблица А.4

Данные таблицы Disrepair

FailureCD	FailureNM
1	Засорилась водогрейная колонка
2	Не горит АГВ
3	Течет из водогрейной колонки
4	Неисправна печная горелка
5	Неисправен газовый счетчик
6	Плохое поступление газа на горелку плиты
7	Туго поворачивается пробка крана плиты
8	При закрытии краника горелка плиты не гаснет
12	Неизвестна

Таблица А.5

Данные таблицы Executor

ExecutorCD	Fio
1	Стародубцев Е. М.
2	Булгаков Т. И.
3	Шубин В. Г.
4	Шлюков М. К.
5	Школьников С. М.
6	Степанов А. В.

Данные таблицы NachislSumma

NachislfactCD	AccountCD	ServiceCD	NachislSum	NachislMonth	NachislYear
1	136160	2	56	1	2018
2	005488	2	46	12	2015
3	005488	2	56	4	2018
4	115705	2	40	1	2015
5	115705	2	250	9	2016
6	136160	1	18,3	1	2017
7	080047	2	80	10	2017
8	080047	2	80	10	2016
9	080270	2	46	12	2016
10	080613	2	56	6	2016
11	115705	2	250	9	2015
12	115705	2	58,7	8	2016
13	136160	2	20	5	2016
15	136169	2	20	5	2016
16	136169	2	58,7	11	2016
17	443069	2	80	9	2016
18	443069	2	38,5	8	2016
19	005488	2	58,7	12	2016
20	015527	1	28,32	7	2017
21	080047	1	19,56	3	2017
22	080613	1	10,6	9	2017
23	443069	1	38,28	12	2017
24	015527	1	38,32	4	2018
25	115705	1	37,15	10	2018
26	080613	1	12,6	8	2015
27	136169	1	25,32	1	2018
28	080270	1	57,1	2	2017
29	136159	1	8,3	8	2018
30	005488	1	62,13	4	2015
31	115705	1	37,8	5	2016
32	443690	1	17,8	6	2017

NachislfactCD	AccountCD	ServiceCD	NachislSum	NachislMonth	NachislYear
33	080047	1	22,56	5	2018
34	126112	1	15,3	8	2015
35	080047	1	32,56	9	2016
36	080613	1	12,6	4	2017
37	115705	1	37,15	11	2018
38	080270	1	58,1	12	2015
39	136169	1	28,32	1	2016
40	015527	1	18,32	2	2017
41	443690	1	21,67	3	2018
42	080613	1	22,86	4	2015
43	080270	1	60,1	5	2016
44	136169	1	28,32	2	2017
45	080047	1	22,2	7	2018
46	126112	1	25,3	8	2016
47	443069	1	38,32	9	2016
48	136159	1	8,3	10	2017
49	115705	1	37,15	6	2018
50	136160	1	18,3	12	2015
51	005488	3	279,8	5	2017
52	005488	3	266,7	2	2018
53	015527	3	343,36	11	2018
54	080047	3	271,6	2	2018
55	080270	3	278,25	11	2018
56	080613	3	254,4	7	2016
57	080613	3	258,8	2	2018
58	080613	3	239,33	5	2018
59	126112	3	179,9	4	2017
60	136159	3	180,13	9	2018
61	136160	3	238,8	3	2015
62	136160	3	237,38	3	2016
63	136169	3	349,19	6	2017
64	136169	3	346,18	7	2017

Окончание табл. А.2

NachislfactCD	AccountCD	ServiceCD	NachislSum	NachislMonth	NachislYear
65	443690	3	290,33	3	2018
66	015527	4	580,1	7	2017
67	015527	4	611,3	10	2018
68	080270	4	444,34	3	2016
69	080270	4	453,43	6	2017
70	080270	4	454,6	4	2018
71	115705	4	553,85	1	2017
72	126112	4	435,5	6	2017
73	136159	4	349,38	4	2016
74	136159	4	418,88	6	2017
75	136169	4	528,44	10	2018
76	443069	4	466,69	5	2017
77	443069	4	444,45	10	2018
78	443690	4	480,88	8	2016
79	443690	4	500,13	9	2017

Таблица А.7

Данные таблицы PaySumma

PayFactCD	AccountCD	ServiceCD	PaySum	PayDate	PayMonth	PayYear
1	005488	2	58,7	08.01.2017	12	2016
2	005488	2	40	06.01.2016	12	2015
3	005488	2	56	06.05.2018	4	2018
4	115705	2	40	10.02.2015	1	2015
5	115705	2	250	03.10.2016	9	2016
6	136160	2	20	13.06.2016	5	2016
7	136160	2	56	12.02.2018	1	2018
8	136169	2	20	22.06.2016	5	2016
9	080047	2	80	26.11.2017	10	2017
10	080047	2	80	21.11.2016	10	2016
11	080270	2	30	03.01.2017	12	2016
12	080613	2	58,5	19.07.2016	6	2016
13	115705	2	250	06.10.2015	9	2015
14	115705	2	58,7	04.09.2016	8	2016

PayFactCD	AccountCD	ServiceCD	PaySum	PayDate	PayMonth	PayYear
15	136169	2	58,7	01.12.2016	11	2016
16	443069	2	80	03.10.2016	9	2016
17	443069	2	38,5	13.09.2016	8	2016
18	136160	1	18	05.02.2017	1	2017
19	015527	1	30	03.08.2017	7	2017
20	080047	1	19,56	02.04.2017	3	2017
21	080613	1	11	03.10.2017	9	2017
22	443069	1	38,28	04.02.2018	12	2017
23	015527	1	40	07.05.2018	4	2018
24	115705	1	37,15	04.11.2018	10	2018
25	080613	1	12	20.09.2015	8	2015
26	136169	1	25,32	03.02.2018	1	2018
27	080270	1	60	05.03.2017	2	2017
28	136159	1	8,3	10.09.2018	8	2018
29	005488	1	65	03.05.2015	4	2015
30	115705	1	37,8	12.07.2016	5	2016
31	443690	1	20	10.07.2017	6	2017
32	080047	1	22,56	25.06.2018	5	2018
33	126112	1	15,3	08.09.2015	8	2015
34	080047	1	32,56	18.10.2016	9	2016
35	080613	1	12,6	22.05.2017	4	2017
36	115705	1	37,15	23.12.2018	11	2018
37	080270	1	58,1	07.01.2016	12	2015
38	136169	1	28,32	08.02.2016	1	2016
39	015527	1	20	18.03.2017	2	2017
40	443690	1	19,47	10.04.2018	3	2018
41	080613	1	22,86	04.05.2015	4	2015
42	080270	1	60	07.06.2016	5	2016
43	136169	1	28,32	05.03.2017	2	2017
44	080047	1	22,2	10.08.2018	7	2018
45	126112	1	25,3	10.09.2016	8	2016
46	443069	1	38,32	09.10.2016	9	2016

PayFactCD	AccountCD	ServiceCD	PaySum	PayDate	PayMonth	PayYear
47	136159	1	8,3	14.11.2017	10	2017
48	115705	1	37,15	10.08.2018	6	2018
49	136160	1	16	07.01.2016	12	2015
50	005488	3	280	10.06.2017	5	2017
51	005488	3	260	11.03.2018	2	2018
52	015527	3	345	15.12.2018	11	2018
53	080047	3	271,6	12.03.2018	2	2018
54	080270	3	278	06.12.2018	11	2018
55	080613	3	254,4	10.08.2016	7	2016
56	080613	3	258,8	08.03.2018	2	2018
57	080613	3	239,35	11.06.2018	5	2018
58	126112	3	179,9	01.05.2017	4	2017
59	136159	3	180,13	21.10.2018	9	2018
60	136160	3	240	04.04.2015	3	2015
61	136160	3	200	06.04.2016	3	2016
62	136169	3	349,19	14.07.2017	6	2017
63	136169	3	346,18	13.08.2017	7	2017
64	443690	3	295	09.04.2018	3	2018
65	015527	4	580,1	08.08.2017	7	2017
66	015527	4	611,3	03.11.2018	10	2018
67	080270	4	444,5	18.04.2016	3	2016
68	080270	4	450	14.07.2017	6	2017
69	080270	4	460	12.05.2018	4	2018
70	115705	4	553,85	02.02.2017	1	2017
71	126112	4	435,5	12.07.2017	6	2017
72	136159	4	349,38	18.05.2016	4	2016
73	136159	4	420	09.07.2017	6	2017
74	136169	4	528,44	26.11.2018	10	2018
75	443069	4	466,69	03.06.2017	5	2017
76	443069	4	444,45	16.11.2018	10	2018
77	443690	4	485	05.09.2016	8	2016

Данные таблицы Request

RequestCD	AccountCD	ExecutorCD	FailureCD	IncomingDate	ExecutionDate	Executed
1	005488	1	1	17.12.2016	20.12.2016	True
2	115705	3	1	07.08.2016	12.08.2016	True
3	015527	1	12	28.02.2017	08.03.2017	False
5	080270	4	1	31.12.2016	NULL	False
6	080613	1	6	16.06.2016	24.06.2016	True
7	080047	3	2	20.10.2017	24.10.2017	True
9	136169	2	1	06.11.2016	08.11.2016	True
10	136159	3	12	01.04.2016	03.04.2016	False
11	136160	1	6	12.01.2018	12.01.2018	True
12	443069	5	2	08.08.2016	10.08.2016	True
13	005488	5	8	04.09.2015	05.12.2015	True
14	005488	4	6	04.04.2018	13.04.2018	True
15	115705	4	5	20.09.2015	23.09.2015	True
16	115705	NULL	3	28.12.2016	NULL	False
17	115705	1	5	15.08.2016	06.09.2016	True
18	115705	2	3	28.12.2017	04.01.2018	True
19	080270	4	8	17.12.2016	27.12.2016	True
20	080047	3	2	11.10.2016	11.10.2016	True
21	443069	1	2	13.09.2016	14.09.2016	True
22	136160	1	7	18.05.2016	25.05.2016	True
23	136169	5	7	07.05.2016	08.05.2016	True

Приложение Б.

Системный каталог базы данных Firebird

Системные таблицы СУБД Firebird содержат метаданные БД. Они формируются автоматически сервером Firebird, когда создается сама БД. Информация, содержащаяся в этих таблицах, определяет типы столбцов таблиц, их названия, связи между таблицами и др. Эти таблицы сопровождаются сервером, и их, конечно, лучше не менять. Однако информацией, содержащейся в системных таблицах, бывает полезно пользоваться, тем более, что для этого можно применять обычные SQL-запросы.

Рассмотрим некоторые системные таблицы и SQL-запросы для получения информации из них [5, 15, 36].

Б.1. Общие сведения о базе данных

Общие сведения о БД СУБД Firebird хранит в системной таблице RDB\$DATABASE, которая всегда содержит только одну строку. Ее сокращенная структура представлена в табл. Б.1.

Таблица Б.1

Структура таблицы RDB\$DATABASE

Имя столбца	Тип	Описание
RDB\$DESCRIPTION	Blob text	Текст комментария к БД. Он сохраняется после резервного копирования и восстановления
RDB\$RELATION_ID	Smallint	Количество таблиц и представлений в БД
RDB\$CHARACTER_SET_NAME	Char (31)	Набор символов по умолчанию для БД

Следует отметить, что поскольку данная таблица всегда содержит только одну строку, то она часто используется в секции FROM запросов, которые возвращают одно вычисляемое значение или контекстную переменную.

Б.2. Домены

Домены (Domains) определяют характеристики столбцов в таблицах БД, т. е. тип поля, значение по умолчанию и пр. Информация о доменах (пользовательских и системных) содержится в таблице RDB\$FIELDS, сокращенная структура которой представлена в табл. Б.2.

Таблица Б.2

Структура таблицы RDB\$FIELDS

Имя столбца	Тип	Описание
RDB\$FIELD_NAME	Char (31)	Для доменов — это имя домена. Для столбцов таблиц и представлений — это уникальное для БД имя поля, связанное с RDB\$FIELD_SOURCE в RDB\$RELATION_FIELDS. Firebird создает домен в этой таблице для каждого определения столбца, которое не наследуется от определенного пользователем домена
RDB\$COMPUTED_SOURCE	Blob text	Текст выражения, которое определяет ограничение домена, вычисляемый столбец
RDB\$FIELD_LENGTH	Smallint	Длина столбца в байтах
RDB\$FIELD_SCALE	Smallint	Отрицательное число задает масштаб для столбцов DECIMAL и NUMERIC
RDB\$FIELD_TYPE	Smallint	Числовой код типа данных для столбца: 7 — SMALLINT, 8 — INTEGER, 12 — DATE, 13 — TIME, 14 — CHAR, 16 — BIGINT, 27 — DOUBLE PRECISION, 35 — TIMESTAMP, 37 — VARCHAR, 261 — BLOB. Коды для DECIMAL и NUMERIC те же, что и для целых типов, используемых для хранения
RDB\$SYSTEM_FLAG	Smallint	Равен 1, если домен был создан для системных таблиц, 0 — для таблиц, определенных пользователем
RDB\$NULL_FLAG	Smallint	Указывает, может ли столбец принимать пустое значение (0) или не может (1)
RDB\$CHARACTER_LENGTH	Smallint	Длина столбца CHAR или VARCHAR в символах

SQL-запрос для получения информации из этой таблицы может выглядеть следующим образом:

```
SELECT RDB$FIELD_NAME
FROM RDB$FIELDS
WHERE RDB$SYSTEM_FLAG = 0 AND
      RDB$FIELD_NAME NOT STARTING WITH 'RDB'
ORDER BY RDB$FIELD_NAME ASC;
```

Этот запрос к учебной БД выбирает имена всех созданных доменов (рис. Б.1).

RDB\$FIELD_NAME
MONEY
PKFIELD
TMONTH
TYEAR

Рис. Б.1. Результат получения имен доменов в учебной БД

Б.3. Отношения (таблицы и представления)

Рассмотрим основные системные таблицы, содержащие описание пользовательских таблиц и представлений. Сокращенная структура таблицы RDB\$RELATIONS, которая хранит сведения об отношениях, представлена в табл. Б.3. Сокращенная структура таблицы RDB\$RELATION_FIELDS, которая содержит сведения о столбцах, представлена в табл. Б.4.

Таблица Б.3

Структура таблицы RDB\$RELATIONS

Имя столбца	Тип	Описание
RDB\$VIEW_ SOURCE	Blob text	Запрос для представления
RDB\$RELATION_ID	Smallint	Внутренний идентификатор таблицы
RDB\$SYSTEM_FLAG	Smallint	Указывает, создана таблица пользователем (0) или системная (1)
RDB\$ FIELD _ID	Smallint	Количество полей в отношении
RDB\$RELATION_NAME	Char (31)	Имя таблицы или представления
RDB\$OWNER_NAME	Varchar (31)	Имя пользователя — владельца (создателя) таблицы или представления

Таблица Б.4

Структура таблицы RDB\$RELATION_FIELDS

Имя столбца	Тип	Описание
RDB\$ FIELD _NAME	Char (31)	Имя столбца
RDB\$RELATION_NAME	Char (31)	Имя таблицы или представления
RDB\$FIELD_ SOURCE	Char (31)	Имя, сгенерированное системой для столбца, коррелирующее с RDB\$FIELDS. Если столбец основан на домене, то хранит имя домена
RDB\$ FIELD _ID	Smallint	Внутренний номер идентификатора
RDB\$SYSTEM_FLAG	Smallint	Определено пользователем (0) или системой (1)
RDB\$DEFAULT_SOURCE	Blob	Значение по умолчанию для столбца
RDB\$GENERATOR_ NAME	Char (31)	Имя внутреннего (автоматически созданного) генератора значений
RDB\$IDENTITY_TYPE	Smallint	Признак столбца-идентификатора

С помощью следующего запроса можно получить имена таблиц и представлений, содержащихся в БД:

```
SELECT RDB$RELATION_NAME
FROM RDB$RELATIONS
WHERE RDB$SYSTEM_FLAG = 0
ORDER BY RDB$RELATION_NAME ASC;
```

Результат выполнения запроса к учебной БД представлен на рис. Б.2.

RDB\$RELATION_NAME
ABONENT
DISREPAIR
EXECUTOR
NACHISLSUMMA
PAYSUMMA
REQUEST
SERVICES
STREET

Рис. Б.2. Результат получения имен таблиц БД

Поскольку в учебной БД не создано пользовательских представлений, то предыдущий запрос возвратил только имена таблиц. Если

бы представления были созданы, то запрос вернул бы список имен таблиц и представлений. Для получения только списка таблиц можно использовать следующий SQL-запрос:

```
SELECT RDB$RELATION_NAME
FROM RDB$RELATIONS
WHERE (RDB$SYSTEM_FLAG = 0) AND (RDB$VIEW_SOURCE IS NULL)
ORDER BY RDB$RELATION_NAME ASC;
```

Все изменения в метаданных, относящихся к таблицам, фиксируются сервером Firebird. Эта информация хранится в системной таблице RDB\$FORMATS.

Каждый раз, когда структура таблицы или представления изменяется, увеличивается номер формата. Когда номер формата любой таблицы достигает 255, вся БД становится недоступной для запросов, пока над ней не будут выполнены операции резервного копирования и восстановления. Сокращенная структура таблицы RDB\$FORMATS представлена в табл. Б.5.

Таблица Б.5

Структура таблицы RDB\$FORMATS

Имя столбца	Тип	Описание
RDB\$RELATION_ID	Smallint	Идентификатор таблицы, представления (такой же, как в RDB\$RELATIONS)
RDB\$FORMAT	Smallint	Идентификатор формата таблицы

Б.4. Процедуры

Сведения о ХП, используемых в БД, находятся в системной таблице RDB\$PROCEDURES. Параметры ХП, входные и выходные, описаны в системной таблице RDB\$PROCEDURE_PARAMETERS. Упрощенная структура таблицы RDB\$PROCEDURES представлена в табл. Б.6, а упрощенная структура таблицы RDB\$PROCEDURE_PARAMETERS — в табл. Б.7.

Таблица Б.6

Структура таблицы RDB\$PROCEDURES

Имя столбца	Тип	Описание
RDB\$PROCEDURE_NAME	Char (31)	Имя процедуры
RDB\$PROCEDURE_ID	Smallint	Идентификатор процедуры
RDB\$PROCEDURE_INPUTS	Smallint	Указывает, существуют входные параметры (1) или нет (0)
RDB\$PROCEDURE_OUTPUTS	Smallint	Указывает, существуют выходные параметры (1) или нет (0)

Имя столбца	Тип	Описание
RDB\$PROCEDURE_SOURCE	Blob text	Исходный код процедуры
RDB\$OWNER_NAME	Varchar (31)	Имя владельца процедуры
RDB\$SYSTEM_FLAG	Smallint	Определена пользователем (0) или системой (1)

Таблица Б.7

Структура таблицы RDB\$PROCEDURE_PARAMETERS

Имя столбца	Тип	Описание
RDB\$PARAMETER_NAME	Char (31)	Имя параметра
RDB\$PROCEDURE_NAME	Char (31)	Имя процедуры
RDB\$PARAMETER_NUMBER	Smallint	Последовательный номер параметра
RDB\$PARAMETER_TYPE	Smallint	Указывает, является параметр входным (0) или выходным (1)
RDB\$FIELD_SOURCE	Char (31)	Сгенерированное системой уникальное имя столбца
RDB\$SYSTEM_FLAG	Smallint	Указывает, определен параметр пользователем (0) или системой (1)

Б.5. Триггеры

Сведения о триггерах, используемых в БД, хранятся в системной таблице RDB\$TRIGGERS. Упрощенная структура данной таблицы представлена в табл. Б.8.

Таблица Б.8

Структура таблицы RDB\$TRIGGERS

Имя столбца	Тип	Описание
RDB\$TRIGGER_NAME	Char (31)	Имя триггера
RDB\$RELATION_NAME	Char (31)	Имя таблицы или представления, для которого используется триггер
RDB\$TRIGGER_SEQUENCE	Smallint	Последовательность (позиция) триггера
RDB\$TRIGGER_TYPE	Smallint	Тип триггера (событие и фаза триггера): 1 — BEFORE INSERT, 2 — AFTER INSERT, 3 — BEFORE UPDATE, 4 — AFTER UPDATE, 5 — BEFORE DELETE, 6 — AFTER DELETE

Имя столбца	Тип	Описание
RDB\$TRIGGER_SOURCE	Blob text	Исходный код триггера
RDB\$TRIGGER_INACTIVE	Smallint	Указывает, является ли триггер активным: 0 —INACTIVE, 1 — ACTIVE
RDB\$SYSTEM_FLAG	Smallint	Определен пользователем (0) или системой (1)

Б.6. Зависимости

Сервер Firebird отслеживает зависимости между объектами БД и хранит их в служебной таблице RDB\$DEPENDENCIES. Упрощенная структура этой таблицы представлена в табл. Б.9.

Таблица Б.9

Структура таблицы RDB\$DEPENDENCIES

Имя столбца	Тип	Описание
RDB\$DEPENDENT_NAME	Char (31)	Имя представления, процедуры, триггера или вычисляемого столбца, на которые происходит ссылка
RDB\$DEPENDED_ON_NAME	Char (31)	Таблица или процедура, на которую ссылается представление, процедура, триггер или вычисляемый столбец
RDB\$FIELD_NAME	Varchar (31)	Имя столбца в таблице зависимости, на который ссылается представление, процедура, триггер или вычисляемый столбец
RDB\$DEPENDENT_TYPE	Smallint	Идентификатор типа объекта (представление, процедура, триггер или вычисляемый столбец). Номер берется из таблицы RDB\$TYPES, объекты нумеруются в соответствии RDB\$FIELD_NAME = 'RDB\$OBJECT_TYPE'
RDB\$DEPENDED_ON_TYPE	Smallint	Идентифицирует тип зависимого объекта

Например, получить список объектов, от которых зависит рассмотренная ранее процедура Factorial, можно с помощью следующего запроса:

```
SELECT d.RDB$DEPENDENT_ON_NAME,  
       d.RDB$DEPENDENT_ON_TYPE,  
       t.RDB$TYPE_NAME  
FROM RDB$DEPENDENCIES d, RDB$TYPES t  
  
WHERE (d.RDB$DEPENDENT_ON_TYPE = t.RDB$TYPE) AND  
       (t.RDB$FIELD_NAME = 'RDB$OBJECT_TYPE') AND  
       (d.RDB$DEPENDENT_NAME = 'Factorial');
```

Результат выполнения запроса представлен на рис. Б.3.

RDB\$DEPENDENT_ON_NAME	RDB\$DEPENDENT_ON_TYPE	RDB\$TYPE_NAME
FACTORIAL	5	PROCEDURE

Рис. Б.3. Результат выбора объектов, от которых зависит процедура Factorial

Из результата запроса следует, что процедура Factorial зависит от самой себя. Это на самом деле так, потому что данная процедура является рекурсивной.

Библиографический список

1. *Пери, Дж.* Введение в Oracle 10g : пер. с англ. / Дж. Пери, Дж. Пост. — М. : ООО «И. Д. Вильямс», 2006.
2. *Moore, S.* Oracle Database Express Edition 2 Day Developer's Guide, 11g Release 2 (11.2) / S. Moore. URL: http://docs.oracle.com/cd/E17781_01/appdev.112/e18147.pdf (дата обращения 30.05.2019).
3. *Lorentz, D.* Oracle Database SQL Language Reference, 11g Release 2 (11.2) / D. Lorentz, M. B. Roeser. URL: https://docs.oracle.com/cd/E11882_01/server.112/e41084/toc.htm (дата обращения 30.05.2019).
4. *Бондарь, А. Г.* Microsoft SQL Server 2014 / А. Г. Бондарь. — СПб. : БХВ-Петербург, 2015.
5. *Борри, Х.* Firebird: руководство разработчика баз данных : пер. с англ. / Х. Борри. — СПб. : БХВ-Петербург, 2006.
6. *Borrie, H.* Firebird 3.0 Release Notes / H. Borrie, 2013.
7. Официальный сайт компании Microsoft. URL: <http://www.microsoft.com> (дата обращения 30.05.2019).
8. *Гринвальд, Р.* Oracle 11g. Основы : пер. с англ. / Р. Гринвальд, Р. Стаковьяк, Дж. Стерн. — 4-е изд. — СПб. : СимволПлюс, 2009.
9. *Виейра, Р.* Программирование баз данных Microsoft SQL Server 2008. Базовый курс : пер. с англ. / Р. Виейра. — М. : ООО «И. Д. Вильямс», 2010.
10. *Алапати, С. Р.* Oracle Database 11g: руководство администратора баз данных. : пер. с англ. / С. Р. Алапати. — М. : Вильямс, 2010.
11. *Murray, C.* Oracle Database Express Edition 2 Day DBA, 11g Release 2 (11.2) / C. Murray. URL: http://docs.oracle.com/cd/E17781_01/server.112/e18804.pdf (дата обращения 30.05.2019).
12. *Lane, P.* Oracle Database Globalization Support Guide, 11g Release 2 (11.2) / P. Lane. URL: https://docs.oracle.com/cd/E11882_01/server.112/e10729/toc.htm (дата обращения 30.05.2019).
13. *Селко, Д.* Стиль программирования Джо Селко на SQL. : пер. с англ. / Д. Селко. — М. : Издательство «Русская редакция»; СПб. : Питер, 2006.
14. Функции по категориям. URL: <http://oracleplsql.ru> (дата обращения 30.11.2018).
15. Руководство по языку SQL СУБД Firebird 3.0 / под ред. Сиимонова Д. URL: http://www.ibase.ru/files/firebird/Firebird_3_0_Language_Reference_RUS.pdf (дата обращения 30.05.2019).

16. Маркин, А. В. Построение запросов и программирование на SQL / А. В. Маркин. — 3-е изд., перераб. и доп. — М. : Издательство «Диалог-МИФИ», 2014.
17. Голицына, О. Л. Базы данных : учеб. пособие / О. Л. Голицына, Н. В. Максимов, И. И. Попов. — М. : Форум: Инфра-М, 2012.
18. Moore, S. Oracle Database PL/SQL Language Reference, 11g Release 2 (11.2) / S. Moore. URL: https://docs.oracle.com/cd/E11882_01/appdev.112/e25519/ (дата обращения 30.05.2019).
19. Официальный сайт компании IBase.ru. URL: <http://www.ibase.ru> (дата обращения 30.05.2019).
20. Словарик по Firebird. URL: <http://www.firebirdsql.su> (дата обращения 29.11.2018).
21. Ковязин, А. Н. Мир InterBase: архитектура, администрирование и разработка приложений баз данных в InterBase/Firebird/Yafill / А. Н. Ковязин, С. М. Востриков. — 4-е изд. — М. : КУДИЦ-ОБРАЗ, 2006.
22. Oracle Pipelined Table Functions. URL: <https://oracle-base.com/articles/misc/pipelined-table-functions> (дата обращения 30.05.2019).
23. Полякова, Л. Основы SQL: электронный курс / Л. Полякова. URL: <http://www.intuit.ru/studies/courses/5/5/info> (дата обращения 30.05.2019).
24. Nanda, A. Efficient PL/SQL Coding // Arup Nanda. Oracle Database 11g: The Top New Features for DBAs and Developers / A. Nanda. URL: <http://www.oracle.com/technetwork/articles/sql/11g-efficient-coding-093640.html> (дата обращения 30.05.2019).
25. Пржиялковский, В. Контекст сеанса в Oracle / В. Пржиялковский. URL: http://citforum.ru/database/oracle/session_context/index.shtml (дата обращения 30.05.2019).
26. Андон, Ф. Язык запросов SQL : учебный курс / Ф. Андон, В. Резниченко. — СПб. : Питер ; Киев : Издательская группа BHV, 2006.
27. Петкович, Д. Microsoft® SQL Server 2012. Руководство для начинающих : пер. с англ. / Д. Петкович. — СПб. : БХВ-Петербург, 2013.
28. Аносов, А. Критерии выбора СУБД при создании информационных систем / А. Аносов. URL: <http://www.interface.ru/home.asp?artId=2147> (дата обращения 30.05.2019).
29. Дейт, К. Дж. Введение в системы баз данных : пер. с англ. / К. Дж. Дейт. — 8-е изд. — М. : Издательский дом «Вильямс», 2005.
30. Грофф, Дж. SQL: полное руководство : пер. с англ. / Дж. Грофф, П. Вайнберг, Э. Оппель. — 3-е изд. — М. : ООО «И. Д. Вильямс», 2015.
31. Логинов, Д. Почему мне нравится Firebird / Д. Логинов. URL: <http://www.loginovprojects.ru/index.php?page=whyfirebird> (дата обращения 30.05.2019).

32. Свидетельство об официальной регистрации разработки в отраслевом фонде алгоритмов и программ. Система дистанционного обучения SQL : № 7044 : 11.11.2006 / П. В. Истомин, А. В. Маркин. — URL: <https://rgrty.ru/sqltest/> (дата обращения 30.05.2019).

33. Свидетельство о государственной регистрации программы для ЭВМ. Система дистанционного обучения «SQLTest» : № 2018611782 : 07.02.2018 / Маркин, А. В., Аникеев, Д. В., Паршин, В. И. — URL: https://rgrty.ru/pluginfile.php/3422/mod_page/content/21/Patent.jpg (дата обращения 30.05.2019).

34. Свидетельство о государственной регистрации программы для ЭВМ. База данных системы дистанционного обучения «SQLTest» : № 2018670015 : 01.06.2018 / Маркин, А. В., Аникеев, Д. В., Паршин, В. И. — URL: https://rgrty.ru/pluginfile.php/3422/mod_page/content/25/2.jpg (дата обращения 30.05.2019).

35. Официальный сайт расчетно-платежного комплекса «Абонент+». URL: <https://www.abonent.plus.ru> (дата обращения 30.05.2019).

36. Бондарь, А. Г. Interbase и Firebird. Практическое руководство для умных пользователей и начинающих разработчиков / А. Г. Бондарь. — СПб. : БХВ-Петербург, 2012.

Новинки по дисциплине

1. *Астапчук, В. А.* Корпоративные информационные системы: требования при проектировании : учебное пособие для вузов / В. А. Астапчук, П. В. Терещенко. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2019. — 113 с. — (Университеты России).

2. *Бессмертный, И. А.* Системы искусственного интеллекта : учебное пособие для академического бакалавриата / И. А. Бессмертный. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2019. — 157 с. — (Бакалавр. Академический курс).

3. *Богатырев, В. А.* Информационные системы и технологии. Теория надежности : учебное пособие для бакалавриата и магистратуры / В. А. Богатырев. — Москва : Издательство Юрайт, 2019. — 318 с. — (Бакалавр и магистр. Модуль).

4. *Гасанов, Э. Э.* Интеллектуальные системы. Теория хранения и поиска информации : учебник для бакалавриата и магистратуры / Э. Э. Гасанов, В. Б. Кудрявцев. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2019. — 271 с. — (Бакалавр и магистр. Академический курс).

5. *Гордеев, С. И.* Организация баз данных в 2 ч. Часть 1 : учебник для вузов / С. И. Гордеев, В. Н. Волошина. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2019. — 310 с. — (Университеты России).

6. *Гордеев, С. И.* Организация баз данных в 2 ч. Часть 2 : учебник для вузов / С. И. Гордеев, В. Н. Волошина. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2019. — 513 с. — (Университеты России).

7. *Грекул, В. И.* Проектирование информационных систем : учебник и практикум для академического бакалавриата / В. И. Грекул, Н. Л. Коровкина, Г. А. Левочкина. — Москва : Издательство Юрайт, 2019. — 385 с. — (Высшее образование).

8. *Григорьев, М. В.* Проектирование информационных систем : учебное пособие для вузов / М. В. Григорьев, И. И. Григорьева. — Москва : Издательство Юрайт, 2019 ; Тюмень : Тюменский государственный университет. — 318 с. — (Высшее образование).

9. *Илюшечкин, В. М.* Основы использования и проектирования баз данных : учебник для академического бакалавриата / В. М. Илюшечкин. — Москва : Издательство Юрайт, 2019. — 213 с. — (Бакалавр. Академический курс).

10. *Кувшинов, Д. Р.* Основы программирования : учебное пособие для вузов / Д. Р. Кувшинов. — Москва : Издательство Юрайт, 2019 ; Екатеринбург : Изд-во Урал. ун-та. — 104 с. — (Университеты России).

11. Методы оптимизации: теория и алгоритмы : учебное пособие для академического бакалавриата / А. А. Черняк, Ж. А. Черняк, Ю. М. Метельский, С. А. Богданович. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2019. — 357 с. — (Бакалавр. Академический курс).

12. *Нестеров, С. А.* Базы данных : учебник и практикум для академического бакалавриата / С. А. Нестеров. — Москва : Издательство Юрайт, 2019. — 230 с. — (Бакалавр. Академический курс).

13. *Нетёсова, О. Ю.* Информационные системы и технологии в экономике : учебное пособие для вузов / О. Ю. Нетёсова. — 3-е изд., испр. и доп. — Москва : Издательство Юрайт, 2019. — 178 с. — (Университеты России).

14. Проектирование информационных систем : учебник и практикум для академического бакалавриата / Д. В. Чистов, П. П. Мельников, А. В. Золотарюк, Н. Б. Ничепорук ; под общей редакцией Д. В. Чистова. — Москва : Издательство Юрайт, 2019. — 258 с. — (Высшее образование).

15. *Рыбальченко, М. В.* Архитектура информационных систем : учебное пособие для вузов / М. В. Рыбальченко. — Москва : Издательство Юрайт, 2019. — 91 с. — (Университеты России).

16. *Стружкин, Н. П.* Базы данных: проектирование : учебник для академического бакалавриата / Н. П. Стружкин, В. В. Годин. — Москва : Издательство Юрайт, 2019. — 477 с. — (Бакалавр. Академический курс).

17. *Стружкин, Н. П.* Базы данных: проектирование. Практикум : учебное пособие для академического бакалавриата / Н. П. Стружкин, В. В. Годин. — Москва : Издательство Юрайт, 2019. — 291 с. — (Бакалавр. Академический курс).