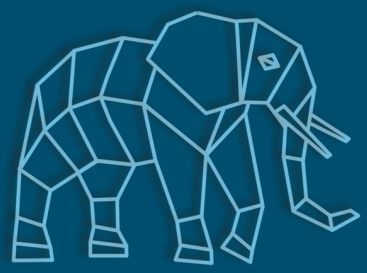


Компания Postgres Professional

Е. П. Моргунов

# ЯЗЫК SQL

# БАЗОВЫЙ КУРС

## УЧЕБНО-ПРАКТИЧЕСКОЕ ПОСОБИЕ

Москва 2017

Компания Postgres Professional

Е. П. Моргунов

# ЯЗЫК SQL. БАЗОВЫЙ КУРС

УЧЕБНО-ПРАКТИЧЕСКОЕ ПОСОБИЕ

Москва 2017

УДК 004.655
ББК 32.973.26-018.2
М79

**Моргунов, Е. П.**

М79 Язык SQL. Базовый курс: учеб.-практ. пособие / Е. П. Моргунов;
под ред. Е. В. Рогова, П. В. Лузанова; Postgres Professional. — М., 2017. — 256 с.

Настоящее учебно-практическое пособие представляет собой первую, базовую, часть учебного курса по языку SQL, предлагаемого российской компанией Postgres Professional. Учебный материал излагается в расчете на использование системы управления базами данных PostgreSQL. Пособие может использоваться как под руководством преподавателя, так и для самостоятельного изучения языка SQL.

Пособие предназначено для студентов, обучающихся по направлениям 09.03.01 – «Информатика и вычислительная техника», 09.03.02 – «Информационные системы и технологии», 09.03.03 – «Прикладная информатика», 09.03.04 – «Программная инженерия» и 02.03.03 – «Математическое обеспечение и администрирование информационных систем». Оно может быть полезно широкому кругу студентов и специалистов, желающих ознакомиться с основами языка SQL в среде системы управления базами данных PostgreSQL.

**УДК 004.655**
**ББК 32.973.26-018.2**

# Оглавление

# Введение

В настоящее время термин «база данных» известен многим людям, даже далеким от профессиональной разработки компьютерных программ. Базы данных стали очень широко распространенной технологией, что потребовало, в свою очередь, большего числа специалистов, способных проектировать их и обслуживать. В ходе эволюции теории и практики баз данных стандартом де-факто стала реляционная модель данных, а в рамках этой модели сформировался и специализированный язык программирования, позволяющий выполнять все необходимые операции с данными — Structured Query Language (SQL). Таким образом, важным компонентом квалификации специалиста в области баз данных является владение языком SQL.

В настоящем учебном пособии излагаются основы языка SQL — это базовый курс. Причем, язык рассматривается применительно к конкретной системе управления базами данных (СУБД) — PostgreSQL. Реализация языка SQL в каждой СУБД соответствует стандарту в той или иной степени, но кроме стандартизированных функций и возможностей, каждая СУБД предлагает и свои дополнительные расширения языка. PostgreSQL обеспечивает очень хорошую поддержку стандарта языка SQL и также предоставляет интересные и практически полезные дополнительные возможности. Одним из главных достоинств PostgreSQL является расширяемость. Это означает, например, что пользователь (конечно, являющийся специалистом в области баз данных) может разработать свои собственные типы данных. Эти типы данных будут обладать всеми свойствами встроенных типов данных и могут быть введены в работу без останова сервера. Кроме того, PostgreSQL является свободно-распространяемым продуктом с открытым исходным кодом, который доступен на большом числе платформ.

В пособии рассматриваются не только все основные команды языка SQL, но также и другие вопросы, такие, как индексы и транзакции.

Пособие написано таким образом, чтобы его можно было использовать как под руководством преподавателя, так и самостоятельно. Предполагается, что студенты имеют доступ к уже установленной СУБД, поэтому процедура установки PostgreSQL детально не рассматривается, а лишь даются указания о том, где найти инструкции по установке.

Это пособие предназначено для получения практических навыков использования языка SQL. Учебный материал подается таким образом, что многие важные знания читатель должен получить в результате выполнения заданий, находящихся в конце каждой главы. В основном тексте глав эти знания могут быть не представлены. Предполагается, что значительная часть заданий будет выполняться читателем самостоятельно с помощью документации на СУБД PostgreSQL, но зачастую даются и указания к их выполнению. Задания, приведенные в пособии, различаются по уровню сложности. Самые сложные из них, а также те, которые требуют много времени для выполнения, отмечены звездочкой.

Задания можно выполнять по мере изучения учебного материала конкретной главы. Однако некоторые из них имеют комплексный характер, поэтому для их выполнения необходимо изучить всю главу или, как минимум, несколько ее разделов.

Хотя пособие имеет практическую направленность и не является теоретическим курсом, все же в первой главе кратко, на элементарном уровне излагаются основные понятия теории баз данных и реляционной модели. Это сделано для того, чтобы студенты могли приступить к практическому освоению языка SQL без задержки, с первых дней учебного семестра, еще до того момента, когда эти понятия будут основательно рассмотрены в лекционном курсе.

На факультетах информационных технологий в российских вузах базы данных традиционно изучаются на втором или третьем курсе. Причем, этой дисциплине, как правило, отводится один семестр. Однако количество академических учебных часов может различаться. Если на практические занятия по этой дисциплине учебный план отводит 36 часов, тогда мы рекомендуем следующее распределение времени на изучение материала пособия.

| | |
|---|---|
| Глава 1. Введение в базы данных и SQL | 1 час |
| Глава 2. Создание рабочей среды | 1 час |
| Глава 3. Основные операции с таблицами | 4 часа |
| Глава 4. Типы данных СУБД PostgreSQL | 4 часа |
| Глава 5. Основы языка определения данных | 4 часа |
| Глава 6. Запросы | 8 часов |
| Глава 7. Изменение данных | 4 часа |
| Глава 8. Индексы | 2 часа |
| Глава 9. Транзакции | 4 часа |
| Глава 10. Повышение производительности | 4 часа |

Главы 1 и 2 могут быть изучены за одно двухчасовое занятие, поскольку СУБД PostgreSQL уже должна быть установлена в учебной аудитории заранее.

Глава 3 представляет собой краткий обзор основных возможностей языка SQL, после ее изучения студенты должны представлять себе простые способы использования всех основных команд языка. Эта глава не очень сложная, но объемная, поэтому на ее изучение отводится четыре часа.

Глава 4 посвящена рассмотрению основных типов данных, используемых в PostgreSQL. Это большая глава, однако в ней значительную часть составляют задания и упражнения. Предполагается, что студенты за четыре часа должны усвоить только основные приемы использования типов данных. А для того, чтобы знания закрепились, рекомендуется обращаться к материалу этой главы (в том числе и к упражнениям) в процессе изучения остальных глав пособия, при необходимости уточнения тех или иных особенностей применения конкретных типов. Распределить время, выделенное на изучение этой главы, мы рекомендуем следующим образом: два часа на первые четыре параграфа — числовые и строковые типы, типы «дата/время» и логический тип, еще два часа — на массивы и тип json/jsonb.

Чтобы выполнять запросы к базе данных, необходимо хорошо понимать ее структуру, взаимосвязи таблиц. Поэтому глава 5, в которой рассматриваются основы языка определения данных, очень важна с точки зрения детального изучения таблиц базы данных «Авиаперевозки» и подготовки к освоению главы 6. Поскольку материал главы основан на том, что база данных уже развернута на компьютере студента, то вводить команды для создания таблиц не требуется. Это позволяет сократить время, затрачиваемое на изучение главы. В пособии принят подход, при котором сначала рассматриваются команды определения данных, а затем — команды манипулирования данными. Поэтому глава 5 «Основы языка определения данных» предшествует

главе 6 «Запросы». Однако избранный подход реализуется не слишком жестко: в обзорной главе 3 рассматриваются основные команды, в том числе, и несложные запросы. А запросы — это уже язык манипулирования данными. На изучение главы 5 отведено четыре часа. В течение первого двухчасового занятия нужно изучить два первых параграфа, в которых освещаются такие вопросы, как ограничения целостности и создание и удаление таблиц. Второе двухчасовое занятие нужно посвятить изучению трех оставшихся параграфов. В них говорится о способах модификации таблиц, а также о представлениях (views) и схемах базы данных.

Глава 6 является центральной главой пособия, поэтому на ее изучение отводится восемь часов, т. е. больше, чем на изучение других глав. Она состоит из четырех параграфов. Первый из них посвящен разнообразным дополнительным возможностям команды SELECT. Речь идет, в частности, о таких вещах, как предложения LIMIT и OFFSET, оператор LIKE и регулярные выражения в условиях предложения WHERE и о других возможностях. Тем не менее, материал этого параграфа несложный, для его изучения достаточно выделить один час. Во втором параграфе рассказывается о способах соединения таблиц. Это более сложная тема, для ее изучения необходимо выделить два часа. Третий параграф посвящен агрегированию и группировке. В нем рассматривается и такая важная и интересная тема, как оконные функции. Данный параграф также требует двухчасового занятия. Самый сложный раздел этой главы — четвертый. Он посвящен подзапросам. В нем, в частности, освещается такая важная и интересная тема, как общие табличные выражения (Common Table Expressions — CTE). Для изучения материала данного параграфа необходимо выделить три часа.

В главе 7 собраны все команды, предназначенные для изменения данных: вставка строк, их обновление и удаление. Поскольку в предшествующих главах эти команды уже использовались для решения простых задач, то в данной главе рассматриваются более сложные способы их использования. В ней много упражнений, они составляют половину ее объема. Рекомендуется уделить два часа изучению способов вставки строк в таблицы, а еще два часа — рассмотрению операций обновления и удаления строк.

Глава 8 посвящена индексам, она небольшая, поэтому с ней можно ознакомиться за одно двухчасовое занятие. Поскольку индексы тесно связаны с вопросами производительности, т. е. скорости выполнения запросов, то было бы целесообразно после изучения заключительной главы вернуться к главе 8 и посмотреть на представленные в ней команды и запросы, уже зная о команде EXPLAIN.

Транзакциям посвящена глава 9. Механизмы их выполнения имеют много тонкостей, поэтому при изучении этой главы необходимо экспериментировать и стараться объяснить полученные результаты.

В заключительной главе 10 рассматриваются вопросы повышения производительности. Эта глава может показаться слишком абстрактной и сложной для начального курса языка SQL, тем не менее, она очень важна. Студенты должны научиться читать планы выполнения запросов и понимать назначение каждой операции, представленной в плане. А овладение искусством оптимизации запросов потребует много времени и опыта, оно придет не сразу.

В том случае, когда на практические занятия по дисциплине «Базы данных» в учебном плане отводится 54 часа, можно изменить предлагаемое распределение учебных часов. В частности, в главе 4 можно больше времени посвятить типам данных

json/jsonb и массивам. В главе 6 можно более детально рассмотреть оконные функции и общие табличные выражения. При изучении главы 9, посвященной транзакциям, было бы целесообразно разработать несложное приложение, в котором использовались бы транзакции, и провести эксперименты с этим приложением, выполняя параллельно несколько сеансов и изменяя при этом уровни изоляции транзакций. В рамках главы 10 имеет смысл вернуться к командам и запросам главы 8 и изучить планы их выполнения с помощью команды EXPLAIN. За счет дополнительного времени можно рассмотреть все задания и упражнения повышенной сложности (помеченные звездочкой).

Таким образом, распределение времени может быть таким:

| | |
|---|---|
| Глава 1. Введение в базы данных и SQL | 1 час |
| Глава 2. Создание рабочей среды | 1 час |
| Глава 3. Основные операции с таблицами | 4 часа |
| Глава 4. Типы данных СУБД PostgreSQL | 6 часов |
| Глава 5. Основы языка определения данных | 6 часов |
| Глава 6. Запросы | 12 часов |
| Глава 7. Изменение данных | 6 часов |
| Глава 8. Индексы | 4 часа |
| Глава 9. Транзакции | 8 часов |
| Глава 10. Повышение производительности | 6 часов |

В пособии используются различные виды шрифтов для выделения фрагментов текста в зависимости от их назначения. Команды, вводимые пользователем как в среде операционной системы, так и в среде утилиты psql, выделяются полужирным моноширинным шрифтом. Например:

```
psql -d demo -U postgres
```

или

```
SELECT avg( total_amount ) FROM bookings;
```

Результаты работы команд операционной системы и SQL-команд, выполняемых в среде утилиты psql, напечатаны моноширинным шрифтом. Например, в ответ на команду

```
EXPLAIN SELECT * FROM aircrafts;
```

на экран будет выведено следующее:

```
                        QUERY PLAN
-----------------------------------------------------------
Seq Scan on aircrafts (cost=0.00..1.09 rows=9 width=52)
(1 строка)
```

Мы надеемся, что изучение материала, изложенного в учебном пособии, будет способствовать повышению уровня вашей квалификации и расширению профессионального кругозора.

# 1 Введение в базы данных и SQL

Эта глава — вводная. В ней мы расскажем об основах баз данных, о том, что такое реляционная модель и зачем нужен язык SQL. Очень важной темой этой главы станет описание предметной области, на основе которой будет спроектирована учебная база данных, которая и будет служить в качестве площадки для изучения языка SQL. Это пособие предназначено в первую очередь для практического освоения языка SQL, а не для изучения теории баз данных, поэтому для изучения теории необходимо обращаться к авторитетным источникам, список которых приведен в конце учебного пособия.

## 1.1 Что такое базы данных и зачем они нужны

Технологии баз данных существовали не всегда. Однако и до их внедрения в практику люди также собирали и обрабатывали данные. Одним из способов хранения данных были так называемые плоские файлы (flat files), которые имели очень простую структуру: данные хранились в виде записей, разделенных на поля фиксированной длины. В реальной жизни между элементами данных зачастую возникают сложные связи, которые необходимо перенести и в электронную базу данных. При использовании плоских файлов эти связи организовать сложно, а еще сложнее поддерживать их при изменениях и удалениях отдельных элементов данных.

Одним из основных понятий в теории баз данных является **модель данных**. Можно сказать, что она характеризует способ организации данных и основные методы доступа к ним. Сначала были предложены иерархическая и сетевая модели данных. Однако в ходе эволюции теорий и идей была разработана реляционная модель данных, которая сейчас и является доминирующей. Поэтому в настоящее время преобладают базы данных реляционного типа. Их характерной чертой является тот факт, что данные воспринимаются пользователем как таблицы. В распоряжении пользователя имеются операторы для выборки данных из таблиц, а также для вставки новых данных, обновления и удаления имеющихся данных.

Одним из достоинств реляционной базы данных является ее способность поддерживать связи между элементами данных, избавляя программиста от необходимости заниматься этой рутинной и очень трудоемкой работой. В те времена, когда технологии реляционных баз данных еще не получили широкого распространения, программистам приходилось на процедурных языках вручную реализовывать такие операции, которые сейчас называются каскадным обновлением внешних ключей или каскадным удалением записей из подчиненных таблиц (файлов). Здесь слово «вручную» означает, что для выполнения этих операций приходилось писать код, состоящий из элементарных команд, позволяющий добраться до каждой обновляемой или удаляемой записи. Тот подход к работе с базами данных назывался навигационным — программист указывал программе конкретный алгоритм поиска записей. Приведем в качестве примера простую ситуацию: в базе данных, построенной на основе файлов, хранится информация о студентах и их экзаменационных оценках, причем, личные данные студентов хранятся в одном файле, назовем его условно «Студенты», а экзаменационные оценки — в другом файле, который условно назовем «Оценки». Если требуется удалить информацию о конкретном студенте и его экзаменационных

оценках, то придется не только выполнить операцию удаления конкретной записи из файла «Студенты», но дополнительно организовать цикл для поиска и удаления тех записей из файла «Оценки», у которых ключевое поле имеет то же значение, что и поле в удаляемой записи из файла «Студенты».

Работая с реляционными базами данных, программист избавлен от программирования на «атомарном» уровне, потому что современные языки для «общения» с этими базами данных являются декларативными. Это означает, что для получения результата достаточно лишь указать, *что* нужно получить, но не требуется предписывать способ получения результата, т. е. *как* его получить.

Система баз данных — это компьютеризированная система, предназначенная для хранения, переработки и выдачи информации по запросу пользователей. Такая система включает в себя программное и аппаратное обеспечение, сами данные, а также пользователей.

Современные системы баз данных являются, как правило, многопользовательскими. В таких системах одновременный доступ к базе данных могут получить сразу несколько пользователей.

Основным программным обеспечением является система управления базами данных. По-английски она называется database management system (DBMS). Кроме СУБД в систему баз данных могут входить утилиты, средства для разработки приложений (программ), средства проектирования базы данных, генераторы отчетов и др.

Пользователи систем с базами данных подразделяются на ряд категорий. Первая категория — это прикладные программисты. Вторая категория — это конечные пользователи, ради которых и выполняется вся работа. Они могут получить доступ к базе данных, используя прикладные программы или универсальные приложения, которые входят в программное обеспечение самой СУБД. В большинстве СУБД есть так называемый **процессор языка запросов**, который позволяет пользователю вводить команды языка высокого уровня (например, языка SQL). Третья категория пользователей — это администраторы базы данных. В их обязанности входят: создание базы данных, выбор оптимальных режимов доступа к ней, разграничение полномочий различных пользователей на доступ к той или иной информации в базе данных, выполнение резервного копирования базы данных и т. д.

Систему баз данных можно разделить на два главных компонента: сервер и набор клиентов (или внешних интерфейсов). Сервер — это и есть СУБД. Клиентами являются различные приложения, написанные прикладными программистами, или встроенные приложения, поставляемые вместе с СУБД. Один сервер может обслуживать много клиентов.

Современные СУБД включают в себя словарь данных. Это часть базы данных, которая описывает сами данные, хранящиеся в ней. Словарь данных помогает СУБД выполнять свои функции.

## 1.2 Основные понятия реляционной модели

В каждой технологической сфере есть своя терминология. Существуют базовые термины, на которых основываются все дальнейшие рассуждения. Такие термины при-

сутствуют и в сфере баз данных. Сейчас мы кратко о них поговорим.

В эпоху, предшествующую рождению реляционной теории, базы данных традиционно рассматривались как набор **файлов**, состоящих из **записей**, а записи, в свою очередь, подразделялись на отдельные **поля**. Поле являлось элементарной единицей данных.

В реляционных базах данных пользователь воспринимает данные в виде таблиц. Поэтому термину «файл» соответствует термин **«таблица»**, вместо термина «запись» используется термин **«строка»**, а вместо термина **«поле»** — термин **«столбец»** (или **«колонка»** ). Таким образом, таблицы состоят из строк и столбцов, на пересечении которых должны находиться «атомарные» значения, которые нельзя разбить на более мелкие элементы без потери смысла.

В формальной теории реляционных баз данных эти таблицы называют **отношениями (relations)** – поэтому и базы данных называются реляционными. Отношение — это математический термин. При определении свойств таких отношений используется теория множеств. В терминах данной теории строки таблицы будут называться **кортежами (tuples)**, а колонки — **атрибутами**. Отношение имеет заголовок, который состоит из атрибутов, и тело, состоящее из кортежей. Количество атрибутов называется **степенью отношения**, а количество кортежей — **кардинальным числом**. Кроме теории множеств, одним из оснований реляционной теории является такой раздел математической логики, как исчисление предикатов.

Таким образом, в теории и практике баз данных существует три группы терминов. Иногда термины из разных групп используют в качестве синонимов, например, запись и строка.

Как мы уже сказали выше, в реляционных базах данных пользователь воспринимает данные в виде таблиц.

Рассмотрим простую систему, в которой всего две таблицы. Первая — «Студенты»:

| № зачетной книжки | Ф. И. О. | Серия документа | Номер документа |
|---|---|---|---|
| 55500 | Иванов Иван Петрович | 0402 | 645327 |
| 55800 | Климов Андрей Иванович | 0402 | 673211 |
| 55865 | Новиков Николай Юрьевич | 0202 | 554390 |

И вторая — «Успеваемость»:

| Зачетная книжка | Предмет | Учебный год | Семестр | Оценка |
|---|---|---|---|---|
| 55500 | Физика | 2016/2017 | 1 | 5 |
| 55500 | Математика | 2016/2017 | 1 | 4 |
| 55800 | Физика | 2016/2017 | 1 | 4 |
| 55800 | Физика | 2016/2017 | 2 | 3 |

При работе с базами данных часто приходится следовать различным **ограничениям**, которые могут быть обусловлены спецификой конкретной предметной области. Упрощая реальную ситуацию, примем следующие ограничения:

– номер зачетной книжки состоит из пяти цифр и не может быть отрицательным (в разных вузах используются различные схемы присваивания номеров зачетным книжкам, эти схемы могут быть гораздо сложнее принятой нами и могут учитывать, например, год поступления студента в вуз);

– серия документа, удостоверяющего личность, является четырехзначным числом, а номер документа, удостоверяющего личность — шестизначным числом;

– номер семестра может принимать только два значения — 1 (осенний семестр) и 2 (весенний семестр);

– оценка может принимать только три значения — 3 (удовлетворительно), 4 (хорошо) и 5 (отлично): другие оценки выставлять в зачетные книжки не принято.

Для идентификации строк в таблицах и для связи таблиц между собой используются так называемые ключи. **Потенциальный ключ** — это комбинация атрибутов таблицы, позволяющая уникальным образом идентифицировать строки в ней. Ключ может состоять и только лишь из одного атрибута таблицы. Например, в таблице «Студенты» таким идентификатором может быть атрибут «Номер зачетной книжки». В качестве потенциального ключа данной таблицы могут также служить два ее атрибута, взятые вместе: «Серия документа, удостоверяющего личность» и «Номер документа, удостоверяющего личность». Ни один из них в отдельности не может использоваться в качестве уникального идентификатора. В таком случае ключ будет составным. При этом важным является то, что потенциальный ключ должен быть *не избыточным*, т. е. никакое подмножество атрибутов, входящих в него, не должно обладать свойством уникальности. Потенциальный ключ, включающий два упомянутых атрибута, является не избыточным.

Ключи нужны для адресации на уровне строк (записей). При наличии в таблице более одного потенциального ключа один из них выбирается в качестве так называемого **первичного ключа**, а остальные будут являться **альтернативными ключами**.

Рассмотрим таблицы «Студенты» и «Успеваемость». Предположим, что в таблице «Студенты» нет строки с номером зачетной книжки 55900, тогда включать строку с таким номером зачетной книжки в таблицу «Успеваемость» не имеет смысла. Таким образом, значения столбца «Номер зачетной книжки» в таблице «Успеваемость» должны быть согласованы со значениями такого же столбца в таблице «Студенты». Атрибут «Номер зачетной книжки» в таблице «Успеваемость» является примером того, что называется **внешним ключом**. Таблица, содержащая внешний ключ, называется **ссылающейся** таблицей (referencing table). Таблица, содержащая соответствующий потенциальный ключ, называется **ссылочной (целевой)** таблицей (referenced table). В таких случаях говорят, что внешний ключ ссылается на потенциальный ключ в ссылочной таблице. Внешний ключ может быть составным, т. е. может включать более одного атрибута. Внешний ключ не обязан быть уникальным. Проблема обеспечения того, чтобы база данных не содержала неверных значений внешних ключей, известна как проблема **ссылочной целостности**. Ограничение, согласно которому значения внешних ключей должны соответствовать значениям потенциальных ключей, называется **ограничением ссылочной целостности (ссылочным ограничением)**.

Обеспечением выполнения ограничений ссылочной целостности занимается СУБД, а от разработчика требуется лишь указать атрибуты, служащие в качестве внешних

ключей. При проектировании баз данных часто предусматривается, что при удалении строки из ссылочной таблицы соответствующие строки из ссылающейся таблицы должны быть также удалены, а при изменении значения столбца, на который ссылается внешний ключ, должны быть изменены значения внешнего ключа в ссылающейся таблице. Этот подход называется **каскадным удалением (обновлением)**.

Иногда применяются и другие подходы. Например, вместо удаления строк из ссылающейся таблицы в этих строках просто заменяют значения атрибутов, входящих во внешний ключ, так называемыми NULL-значениями. Это специальные значения, означающие «ничто» или отсутствие значения, они не совпадают со значением «нуль» или «пустая строка». NULL-значение применяется в базах данных и в качестве значения по умолчанию, когда пользователь не ввел никакого конкретного значения. Первичные ключи не могут содержать NULL-значений.

Транзакция — одно из важнейших понятий теории баз данных. Она означает набор операций над базой данных, рассматриваемых как единая и неделимая единица работы, выполняемая полностью или не выполняемая вовсе, если произошел какой-то сбой в процессе выполнения транзакции. Таким образом, транзакции являются средством обеспечения согласованности данных. В нашей базе данных транзакцией могут быть, например, две операции: удаление строки из таблицы «Студенты» и удаление связанных по внешнему ключу строк из таблицы «Успеваемость».

## 1.3 Что такое язык SQL

Язык SQL — это непроцедурный язык, который является стандартным средством работы с данными во всех реляционных СУБД. Операторы (команды), написанные на этом языке, лишь указывают СУБД, какой результат должен быть получен, но не описывают процедуру получения этого результата. СУБД сама определяет способ выполнения команды пользователя. В языке SQL традиционно выделяются группа операторов определения данных (Data Definition Language — DDL), группа операторов манипулирования данными (Data Manipulation Language — DML) и группа операторов, управляющих привилегиями доступа к объектам базы данных (Data Control Language — DCL).

К операторам языка определения данных (DDL) относятся команды для создания, изменения и удаления таблиц, представлений и других объектов базы данных. Детальному рассмотрению этих команд посвящены главы 5 и 8.

К операторам языка манипулирования данными (DML) относятся команды для выборки строк из таблиц, вставки строк в таблицы, обновления и удаления строк. Эти команды подробно рассматриваются в главах 6 и 7.

Операторы DCL в пособии не рассматриваются, т. к. PostgreSQL позволяет на начальном этапе изучения языка SQL обойтись без их использования.

## 1.4 Описание предметной области и учебной базы данных

Чтобы показать все основные возможности языка SQL, нам потребуется база данных. Эта база данных не должна быть слишком сложной, чтобы ее изучение не потребовало слишком много времени. Но, вместе с тем, она должна быть достаточно разнообразной, чтобы запросы к ней выглядели бы правдоподобными, почти такими же, как и в реальной работе.

В качестве предметной области выберем пассажирские авиаперевозки. Ее оригинальное описание и описание базы данных «Авиаперевозки» можно найти по адресам `https://postgrespro.ru/education/demodb` и `https://postgrespro.ru/docs/postgrespro/current/demodb-bookings.html`. Надеемся, что эта область знакома многим читателям нашего учебного пособия. Конечно, в учебных целях реальная ситуация намеренно упрощена, но все принципиальные вещи сохранены.

Итак, некая российская авиакомпания выполняет пассажирские авиаперевозки. Она обладает своим парком самолетов различных моделей. Каждая модель самолета имеет определенный код, который присваивает Международная ассоциация авиаперевозчиков (IATA). При этом будем считать, что самолеты одной модели имеют одинаковые компоновки салонов, т. е. порядок размещения кресел и нумерацию мест в салонах бизнес-класса и экономического класса. Например, если это модель Sukhoi SuperJet-100, то место 2A относится к бизнес-классу, а место 20D — к экономическому классу. Бизнес-класс и экономический класс — это разновидности так называемого класса обслуживания.

Наша авиакомпания выполняет полеты между аэропортами России. Каждому аэропорту присвоен уникальный трехбуквенный код, при этом используются только заглавные буквы латинского алфавита. Эти коды присваивает не сама авиакомпания, а специальные организации, управляющие пассажирскими авиаперевозками. Зачастую название аэропорта не совпадает с названием того города, которому этот аэропорт принадлежит. Например, в городе Новосибирске аэропорт называется Толмачево, в городе Екатеринбурге — Кольцово, а в Санкт-Петербурге — Пулково. К тому же некоторые города имеют более одного аэропорта. Сразу в качестве примера вспоминается Москва с ее аэропортами Домодедово, Шереметьево и Внуково. Добавим еще одну важную деталь: каждый аэропорт характеризуется географическими координатами — долготой и широтой, а также часовым поясом.

Формируются маршруты перелетов между городами. Конечно, каждый такой маршрут требует указания не только города, но и аэропорта, поскольку, как мы уже сказали, в городе может быть и более одного аэропорта. В качестве упрощения реальности мы решим, что маршруты не будут иметь промежуточных посадок, т. е. у них будет только аэропорт отправления и аэропорт назначения. Каждый маршрут имеет шестизначный номер, включающий цифры и буквы латинского алфавита.

На основе перечня маршрутов формируется расписание полетов (или рейсов). В расписании указывается плановое время отправления и плановое время прибытия, а также тип самолета, выполняющего этот рейс.

При фактическом выполнении рейса возникает необходимость в учете дополнительных сведений, а именно: фактического времени отправления и фактического времени прибытия, а также статуса рейса. Статус рейса может принимать ряд значений:

– Scheduled (за месяц открывается возможность бронирования);
– On Time (за сутки открывается регистрация);
– Delayed (рейс задержан);
– Departed (вылетел);
– Arrived (прибыл);
– Cancelled (отменен).

Теперь обратимся к пассажирам. Полет начинается с бронирования авиабилета. В настоящее время общепринятой практикой является оформление электронных билетов. Каждый такой билет имеет уникальный номер, состоящий из 13 цифр. В рамках одной процедуры бронирования может быть оформлено несколько билетов, но каждая такая процедура имеет уникальный шестизначный номер (шифр) бронирования, состоящий из заглавных букв латинского алфавита и цифр. Кроме того, для каждой процедуры бронирования записывается дата бронирования и рассчитывается общая стоимость оформленных билетов.

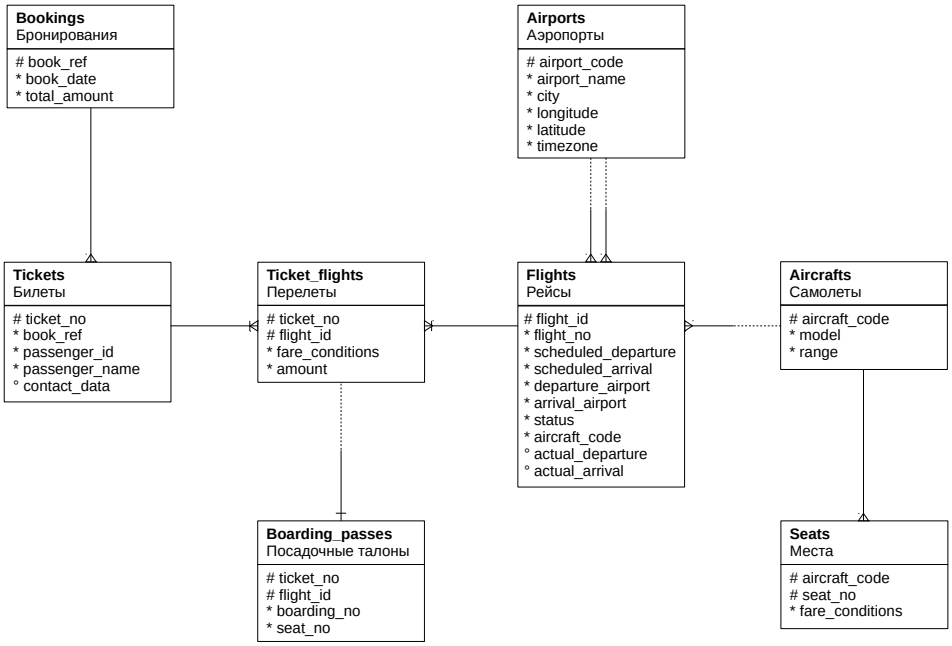
В каждый билет, кроме его тринадцатизначного номера, записывается идентификатор пассажира, а также его имя и фамилия (в латинской транскрипции) и контактные данные. В качестве идентификатора пассажира используется номер документа, удостоверяющего личность. Конечно, пассажир может сменить свой документ, а иной раз даже фамилию и имя, за время, прошедшее между бронированием билетов в разные дни, поэтому невозможно наверняка сказать, что какие-то конкретные билеты были оформлены на одного и того же пассажира.

В каждый электронный билет может быть вписано более одного перелета. Специалисты называют эти записи о перелетах сегментами. В качестве примера наличия нескольких сегментов можно привести такой: Красноярск — Москва, Москва — Анапа, Анапа — Москва, Москва — Красноярск. При этом возможно в рамках одного бронирования оформить несколько билетов на различных пассажиров. Для каждого перелета указывается номер рейса, аэропорты отправления и назначения, время вылета и время прибытия, а также стоимость перелета. Кроме того, указывается и так называемый класс обслуживания: экономический, бизнес и др.

Когда пассажир прибывает в аэропорт отправления и проходит регистрацию билета, оформляется так называемый посадочный талон. Этот талон связан с авиабилетом: в талоне указывается такой же номер, который имеет электронный авиабилет данного пассажира. Кроме того, в талоне указывается номер рейса и номер места в самолете. Указывается также и номер посадочного талона — последовательный номер, присваиваемый в процессе регистрации билетов на данный рейс.

Напомним, что каждому креслу в салоне самолета соответствует конкретный класс обслуживания. Данная информация учитывается при регистрации билетов и оформлении посадочных талонов. Если, например, пассажир приобрел билет с экономическим классом обслуживания, то в его посадочном талоне будет указан номер места в салоне экономического класса, но не в салоне бизнес-класса.

Следуя приведенному описанию предметной области, можно спроектировать модельную базу данных, предназначенную для изучения языка SQL. Поскольку наше учебное пособие в первую очередь предназначено для изучения языка SQL, а не для овладения искусством проектирования баз данных, то мы приведем лишь диаграмму




схемы данных, на которой показаны сущности, выделенные в предметной области, а также их связи и атрибуты. Конкретные же типы данных, первичные и внешние ключи, а также ограничения, наложенные на атрибуты и таблицы, мы покажем уже в последующих главах в процессе рассмотрения команд SQL, предназначенных для физического создания таблиц в базе данных.

Приведенную схему можно найти на сайте компании Postgres Professional по адресам, указанным в начале этого раздела.

## Контрольные вопросы и задания

1. Какие группы операторов выделяются в составе языка SQL?
2. Дайте неформальное определение основных понятий реляционной модели данных: отношение, кортеж, атрибут.
3. Для чего нужны внешние ключи в реляционных таблицах?
4. Что такое потенциальный ключ?
5.* Предложите пример избыточного потенциального ключа для одной из таблиц базы данных «Авиаперевозки» и объясните, почему он будет избыточным.

6.* В текущей реализации базы данных «Авиаперевозки» предполагается, что самолеты одной модели могут иметь только одну компоновку салона. Представим, что руководством принято решение о том, что нужно учитывать возможность наличия различных компоновок для каждой модели. Какие таблицы придется модифицировать в таком случае и каким образом? Потребуется ли создавать дополнительные таблицы?

# 2 Создание рабочей среды

Прежде чем приступать к непосредственному изучению языка SQL, нужно получить доступ к серверу PostgreSQL. Это можно сделать, например, в компьютерном классе или путем обращения к удаленному серверу через терминал. Однако можно создать рабочую среду для себя и на своем локальном компьютере, установив полную версию СУБД PostgreSQL, т. е. сервер и клиентские программы. В этом случае у вас будет гораздо больше полномочий по настройке и использованию PostgreSQL.

В заключительной части главы мы покажем, как развернуть учебную базу данных «Авиаперевозки», наполненную специально подготовленными правдоподобными данными.

## 2.1 Установка СУБД

Поскольку настоящее учебное пособие предназначено для изучения языка SQL, а не основ администрирования СУБД PostgreSQL, то мы ограничимся лишь краткими указаниями о том, где найти инструкции по установке.

Начать нужно с выбора того дистрибутива СУБД, который вы хотели бы установить. Вы можете выбрать оригинальный вариант PostgreSQL или тот, который предлагается компанией Postgres Professional. Он называется Postgres Pro и содержит не только все функции и модули, входящие в состав стандартного дистрибутива, но и дополнительные разработки, выполненные в компании Postgres Professional. Для изучения основ языка SQL эти дистрибутивы подходят в равной степени. Однако документация на русском языке включена только в состав Postgres Pro.

После того как вы определитесь с конкретным дистрибутивом СУБД, необходимо выбрать операционную систему. PostgreSQL поддерживает множество систем, в том числе различные версии Linux, а также Windows.

Устанавливать рекомендуется последнюю **стабильную** версию СУБД.

Если вы решили воспользоваться оригинальным дистрибутивом PostgreSQL, то найти инструкции по его установке в различных операционных системах можно по адресу `https://www.postgresql.org/download/`.

Если же вы остановили свой выбор на дистрибутиве Postgres Pro, тогда следует обратиться сюда: `https://postgrespro.ru/products/postgrespro/download/latest`.

После установки как PostgreSQL, так и Postgres Pro в среде Windows придется предпринять дополнительные меры, чтобы использование русского алфавита в интерактивном терминале psql не вызывало проблем. Утилита psql рассматривается в следующем разделе.

В процессе установки будет создана учетная запись пользователя СУБД с именем postgres. Для изучения настоящего пособия создавать дополнительные учетные записи не требуется.

Установив тот или иной дистрибутив PostgreSQL, нужно научиться запускать сервер баз данных, потому что иначе невозможно работать с данными. Как это сделать, подробно описано в документации в разделе 18.3 «Запуск сервера баз данных». Найти этот раздел можно по адресу `https://postgrespro.ru/docs/postgresql/current/server-start.html`. При установке СУБД в среде Windows создается служба (service) для автоматического запуска сервера PostgreSQL при загрузке операционной системы.

Завершив работу с сервером, нужно корректно остановить (выключить) его. Порядок действий в такой ситуации описан в документации в разделе 18.5 «Выключение сервера». Найти этот раздел можно по адресу `https://postgrespro.ru/docs/postgresql/9.6/server-shutdown.html`.

## 2.2 Программа psql — интерактивный терминал PostgreSQL

Для доступа к серверу баз данных в комплект PostgreSQL входит интерактивный терминал psql. Для его запуска нужно ввести команду

```
psql
```

При запуске утилиты psql в среде Windows возможно некорректное отображение букв русского алфавита. Для устранения этого потребуется в свойствах окна, в котором выполняется psql, изменить шрифт на Lucida Console и с помощью команды chcp сменить текущую кодовую страницу на CP1251:

```
chcp 1251
```

В среде утилиты psql можно вводить не только команды языка SQL, но и различные сервисные команды, поддерживаемые самой утилитой. Для получения краткой справки по всем сервисным командам нужно ввести

```
\?
```

Многие такие команды начинаются с символов \d. Например, для того чтобы просмотреть список всех таблиц и представлений (views), созданных в той базе данных, к которой вы сейчас подключены, введите команду

```
\dt
```

Если же вас интересует определение (попросту говоря, структура) какой-либо конкретной таблицы базы данных, например, students, нужно ввести команду

```
\d students
```

Для получения списка всех SQL-команд нужно выполнить команду

```
\h
```

Для вывода описания конкретной SQL-команды, например, CREATE TABLE, нужно сделать так:

```
\h CREATE TABLE
```

Эта утилита позволяет сокращать объем ручного ввода за счет дополнения вводимой команды «силами» psql. Например, при вводе SQL-команды можно использовать клавишу Tab для дополнения вводимого ключевого слова команды или имени таблицы базы данных. Например, при вводе команды CREATE TABLE . . . можно, введя символы «cr», нажать клавишу Tab — psql дополнит это слово до «create». Аналогично можно поступить и со словом TABLE. Для его ввода достаточно ввести лишь буквы «ta» и нажать клавишу Tab. Если вы ввели слишком мало букв для того, чтобы утилита psql могла однозначно идентифицировать ключевое слово, дополнения не произойдет. Но в таком случае вы можете нажать клавишу Tab дважды и получить список всех ключевых слов, начинающихся с введенной вами комбинации букв.

## 2.3 Развертывание учебной базы данных

Завершив установку сервера баз данных, мы можем перейти непосредственно к рассмотрению вопроса о том, как развернуть в вашем кластере PostgreSQL учебную базу данных «Авиаперевозки», подготовленную компанией Postgres Professional.

На сайте компании есть раздел, посвященный этой базе данных, найти его можно по ссылке `https://postgrespro.ru/education/demodb`. Она предоставляется в трех версиях, отличающихся только объемом данных: самая компактная версия содержит данные за один месяц, версия среднего размера охватывает временной период в три месяца, а самая полная версия включает данные за целый год. Все данные были сгенерированы с помощью специальных алгоритмов, обеспечивающих их «правдоподобность». Мы рекомендуем вам начать с компактной версии базы данных «Авиаперевозки», а после получения некоторого опыта написания SQL-запросов вы установите полную версию и уже на ней сможете лучше «прочувствовать» различные тонкости работы с данными больших объемов, например, оцените влияние индексов на скорость доступа к данным.

В качестве первого шага к развертыванию базы данных нужно скачать ее заархивированную резервную копию по ссылке `https://edu.postgrespro.ru/demo_small.zip`. Затем необходимо извлечь файл из архива:

```
unzip demo_small.zip
```

Извлеченный файл называется demo_small.sql. Теперь мы создадим базу данных с именем demo в вашем кластере PostgreSQL. Самый краткий вариант команды будет таким:

```
psql -f demo_small.sql -U postgres
```

Если вы хотите перенаправить вывод сообщений, которые генерирует СУБД в процессе работы, с экрана в файлы, то можно поступить так:

```
psql -f demo_small.sql -U postgres > demo.log 2>demo.err
```

Можно разделить стандартное устройство вывода и стандартное устройство вывода ошибок. Обычные сообщения будут перенаправлены в файл demo.log, а сообщения об ошибках — в файл demo.err. Обратите внимание, что между цифрой 2, обозначающей дескриптор стандартного устройства вывода сообщений об ошибках, и знаком «>», обозначающим переадресацию вывода, не должно быть пробела.

Если вам удобнее собрать все сообщения в один общий файл, тогда нужно сделать так:

```
psql -f demo_small.sql -U postgres > demo.log 2>&1
```

Обратите внимание, что все выражение 2>&1 в конце команды пишется без пробелов. Оно указывает операционной системе, что сообщения об ошибках нужно направить туда же, куда выводятся и обычные сообщения.

Если бы наш SQL-файл был очень большим, тогда можно было бы выполнить команду в фоновом режиме, поставив в конце командной строки символ «&», а за ходом процесса в реальном времени наблюдать с помощью команды tail.

```
psql -f demo_small.sql -U postgres > demo.log 2>&1 &
tail -f demo.log
```

Выберите один из предложенных вариантов команды для развертывания базы данных и выполните эту команду.

Все готово! Можно подключаться к новой базе данных:

```
psql -d demo -U postgres
```

## Контрольные вопросы и задания

1. Выполните процедуру установки СУБД PostgreSQL в среде выбранной вами операционной системы.
2. Ознакомьтесь с утилитой psql с помощью встроенной справки, а также с помощью справки, вызываемой по команде

   ```
   psql --help
   ```

3. Кроме утилиты psql существуют и другие универсальные программы для работы с сервером баз данных PostgreSQL, например, pgAdmin. Это мощная утилита с графическим интерфейсом.

   Самостоятельно установите программу pgAdmin и изучите основные приемы работы с ней.
4. Выполните развертывание учебной базы данных. Попробуйте подключиться к ней с помощью утилиты psql. Для выхода из утилиты используйте команду \q.

# 3 Основные операции с таблицами

Язык SQL — очень многообразный, он включает в себя целый ряд команд, которые, в свою очередь, иной раз имеют множество параметров и ключевых слов. Но начнем мы с краткого обзора основных возможностей языка SQL. В этой главе вы научитесь вводить данные в базу данных, освоите основные способы получения информации из базы данных, т. е. выборки, а также узнаете, как можно внести изменения в информацию, хранящуюся в базе данных, и удалить те данные, которые больше не нужны.

В практике изучения иностранных языков есть хорошая традиция. Уже на первом занятии ученик изучает некоторые базовые грамматические конструкции и слова, позволяющие ему сказать несколько самых простых, но, тем не менее, практически полезных фраз. Мы последуем этой традиции. В данном разделе нашего пособия вы ознакомитесь с основными командами языка SQL, которые позволят вам выполнять базовые операции. А более сложные (и интересные) команды вы изучите в следующих главах.

Скажем два слова о нашем подходе к работе. В принципе возможны два способа организации работы студента (обучающегося). Первый способ таков: студент использует базу данных, в которой уже содержатся все необходимые таблицы и другие объекты базы данных, подготовленные заранее автором учебного пособия или другим квалифицированным специалистом. При этом некоторый набор необходимых данных также уже введен в таблицы, поэтому можно сразу же переходить к выполнению запросов к этим таблицам. Описанный способ кажется очень привлекательным, поскольку он требует меньше усилий на начальном этапе освоения языка SQL. Однако, на наш взгляд, более правильным является другой способ. Наверное, он более трудоемкий, но при его использовании вы лучше, как говорится, прочувствуете процесс создания таблиц и ввода записей в эти таблицы. А выполняя различные запросы к базе данных, вам будет легче оценить правильность полученного результата выполнения запроса, поскольку вы ввели все данные самостоятельно и поэтому сможете обоснованно предположить, какие результаты ожидаете увидеть на экране. Конечно, первый способ может быть очень полезным при изучении более сложных, продвинутых, возможностей языка SQL, которые трудно понять без использования больших массивов данных, а большие массивы данных вводить в базу данных вручную — нерационально. Гораздо более рациональным будет их автоматическое формирование программным путем.

В главе 1 мы описали предметную область, поэтому сейчас можем приступить к непосредственному созданию таблиц в базе данных. Для выполнения всех последующих команд и операций мы будем использовать утилиту psql, входящую в стандартную поставку СУБД PostgreSQL.

На вашем компьютере уже должна быть развернута база данных demo. Процесс ее создания описан в главе 2. Теперь запустите утилиту psql и подключитесь к этой базе данных с учетной записью пользователя postgres:

```
psql -d demo --U postgres
```

Для создания таблиц в языке SQL служит команда CREATE TABLE. Ее полный синтаксис представлен в документации на PostgreSQL, а упрощенный синтаксис таков:

```
CREATE TABLE "имя_таблицы"
( имя_поля тип_данных [ограничения_целостности],
  имя_поля тип_данных [ограничения_целостности],
...
  имя_поля тип_данных [ограничения_целостности],
  [ограничение_целостности],
  [первичный_ключ],
  [внешний_ключ]
);
```

В квадратных скобках показаны необязательные элементы команды. После команды нужно поставить символ «;».

Для получения в среде утилиты psql полной информации о команде CREATE TABLE сделайте так:

```
\h CREATE TABLE
```

Обратите внимание на отсутствие символа «;» в конце строки.

Наименование SQL-команды можно вводить и в нижнем регистре, т. е. строчными буквами:

```
\h create table
```

В качестве первой таблицы, которую мы создадим, выберем «Самолеты». Таблица имеет следующую структуру (т. е. набор атрибутов и их типы данных):

| Описание атрибута | Имя атрибута | Тип данных | Тип PostgreSQL | Ограничения |
|---|---|---|---|---|
| Код самолета, IATA | aircraft_code | Символьный | char(3) | NOT NULL |
| Модель самолета | model | Символьный | text | NOT NULL |
| Максимальная дальность полета, км | range | Числовой | integer | NOT NULL<br>range > 0 |

Типы char и text являются символьными типами данных и позволяют вводить любые символы, в том числе буквы и цифры. Для атрибута «Код самолета, IATA» мы выбрали тип char(3), поскольку эти коды состоят из трех символов: букв и цифр. Число 3 в описании типа данных char означает максимальное количество символов, которые можно ввести в это поле.

Наименования конкретных моделей самолетов могут содержать различные количества разных символов, поэтому для атрибута «Модель самолета» мы выбрали тип данных text, который не требует указания максимальной длины сохраняемого значения. Вообще, число символов, которые можно сохранить в поле типа text, практически не ограничено.

Для атрибута «Максимальная дальность полета» мы выбрали целый числовой тип. Значения всех атрибутов каждой строки данной таблицы должны быть определенными, поэтому на них накладывается ограничение NOT NULL. В принципе в таблицах базы данных могут содержаться неопределенные значения некоторых атрибутов. Говоря другими словами, их значения могут отсутствовать. В таких случаях в этих полях содержится специальное значение NULL. Но в таблице «Самолеты» не допускается отсутствие значений атрибутов, отсюда и возникает ограничение NOT NULL. К тому же атрибут «Максимальная дальность полета» не должен принимать отрицательных

значений и нулевого значения, поэтому приходится добавить еще одно ограничение: range > 0.

В качестве первичного ключа выбран атрибут «Код самолета, IATA». Таким образом, первичный ключ будет, как говорят, **естественным**. Это означает, что и в реальной предметной области существует такое понятие, как код самолета, и это понятие используется на практике. В отличие от естественных ключей иногда используются и так называемые **суррогатные ключи**, но о них мы расскажем в последующих главах пособия.

Итак, команда для создания нашей первой таблицы «Самолеты» такова:

```
CREATE TABLE aircrafts
( aircraft_code char( 3 ) NOT NULL,
  model text NOT NULL,
  range integer NOT NULL,
  CHECK ( range > 0 ),
  PRIMARY KEY ( aircraft_code )
);
```

Прежде чем вы сможете приступить к непосредственному вводу этой команды в командной строке утилиты psql, мы дадим ряд рекомендаций.

Для СУБД регистр символов (прописные или строчные буквы), используемых для ввода ключевых (зарезервированных) слов, значения не имеет. Однако традиционно ключевые слова языка SQL вводят в верхнем регистре, что повышает наглядность SQL-операторов. Тем не менее, наименования типов данных (integer, char, text и т. д.) мы будем писать не заглавными буквами, а строчными, поскольку именно так «поступает» утилита pg_dump (входящая в комплект поставки PostgreSQL), которая предназначена для создания резервной копии базы данных. Конечно, при выполнении заданий, приводимых в нашем учебном пособии, допустимо для ускорения набора вводить в нижнем регистре и ключевые слова. А в реальной работе нужно следовать тем правилам оформления исходных кодов, которые приняты в рамках выполняемого проекта.

Эту команду для создания таблицы aircrafts (как и все SQL-команды) в утилите psql можно вводить двумя способами. Первый способ заключается в том, что команда вводится полностью на одной строке, при этом строка сворачивается «змейкой». Нажимать клавишу Enter после ввода каждого фрагмента команды не нужно, но можно для повышения наглядности вводить пробел. На экране это выглядит так:

```
demo=# CREATE TABLE aircrafts ( aircraft_code char( 3 ) NOT NULL, model
text NOT NULL, range integer NOT NULL, CHECK ( range > 0 ), PRIMARY KEY
( aircraft_code ) );
```

Второй способ заключается в построчном вводе команды точно так же, как она напечатана в тексте главы. При этом после ввода каждой строки нужно нажимать клавишу Enter. Обратите внимание, что до тех пор, пока команда не введена полностью, вид приглашения к вводу команд, выводимого утилитой psql, будет отличаться от первоначального. В конце команды необходимо поставить точку с запятой.

```
demo=# CREATE TABLE aircrafts
demo-# ( aircraft_code char( 3 ) NOT NULL,
demo(# model text NOT NULL,
```

```
demo(# range integer NOT NULL,
demo(# CHECK ( range > 0 ),
demo(# PRIMARY KEY ( aircraft_code )
demo(# );
```

В среде утилиты psql предлагаются и другие способы завершения вводимых команд с целью их последующего выполнения. Например, вместо ввода символа «;» команду можно завершить символами «\g»:

```
demo=# CREATE TABLE aircrafts ... \g
```

Впоследствии можно с помощью клавиши «стрелка вверх» вызвать на экран (из буфера истории введенных команд) всю команду полностью в компактном виде и при необходимости отредактировать ее либо выполнить еще раз без редактирования. При этом для команды, введенной построчно, сохраняется ее построчная структура, а приглашение выводится только для первой строки:

```
demo=# CREATE TABLE aircrafts
( aircraft_code char( 3 ) NOT NULL,
  model text NOT NULL,
  range integer NOT NULL,
  CHECK ( range > 0 ),
  PRIMARY KEY ( aircraft_code )
);
```

Для перемещения курсора по «виртуальным» строкам команды при ее редактировании нужно использовать клавиши «стрелка влево» и «стрелка вправо», но не «стрелка вверх» или «стрелка вниз».

Если вы хотите непосредственно из среды psql вызвать внешний редактор для редактирования текущего буфера запроса, то нужно воспользоваться командой \e.

Если вы решили прервать ввод команды, еще не введя ее полностью, то просто нажмите клавиши Ctrl-C, в результате ввод команды будет прерван, а приглашение к вводу, выводимое утилитой psql, примет свой первоначальный вид:

```
demo=# CREATE TABLE aircrafts
( aircraft_code char( 3 ) NOT NULL,
demo(# ^C
demo=#
```

Теперь выберите способ ввода команды для создания таблицы aircrafts и введите ее. Если вы не допустили ошибок, то в ответ psql выведет сообщение, означающее успешное создание таблицы:

```
CREATE TABLE
```

Вы можете проверить, какую таблицу создала СУБД. Для этого служит команда утилиты psql

```
\d aircrafts
```

В ответ вы получите примерно такой вывод на экран:

```
Таблица "public.aircrafts"
    Колонка     |     Тип      | Модификаторы
----------------+--------------+--------------
 aircraft_code  | character(3) | NOT NULL
 model          | text         | NOT NULL
 range          | integer      | NOT NULL
Индексы:
 "aircrafts_pkey" PRIMARY KEY, btree (aircraft_code)
Ограничения-проверки:
 "aircrafts_range_check" CHECK (range > 0)
```

В этом выводе новым для вас может быть выражение «public.aircrafts». В нем слово public означает имя так называемой **схемы**. Это, упрощенно говоря, раздел базы данных, в котором и создаются таблицы и другие объекты. По умолчанию используется схема public. О схемах мы будем говорить более подробно в последующих главах пособия.

В описание таблицы входит также информация о созданных индексах. Индекс — это специальная структура данных, позволяющая решать задачу ускорения доступа к строкам в таблице, а также задачу предотвращения дублирования значений ключевых атрибутов в различных строках таблицы. Для реализации первичного ключа (PRIMARY KEY) всегда автоматически создается индекс. Имя индекса в наше случае — aircrafts_pkey. Оно было сгенерировано ядром PostgreSQL. Указан также и тип индекса — btree, т. е. B-дерево. Далее в круглых скобках приводится список ключевых атрибутов. В нашем случае он состоит из одного атрибута — aircraft_code.

Далее в описании таблицы приводятся сведения об ограничениях, наложенных на отдельные атрибуты таблицы и на таблицу в целом. В принципе, при создании таблицы можно задать свои собственные имена для всех ограничений, однако делать это не обязательно. Мы не задавали никакого имени для ограничения, наложенного на атрибут range, поэтому ядро PostgreSQL также сгенерировало это имя автоматически — aircrafts_range_check.

Следует различать команды языка SQL и команды утилиты psql. Команды, начинающиеся с символа «\» являются командами, которые утилита psql предлагает для удобства пользователя.

Поскольку таблицы, которые мы будем сейчас создавать, очень простые, то в случае выявления какого-либо упущения при их создании, вы можете просто удалить таблицу и создать ее заново, с учетом необходимых исправлений. А команду ALTER TABLE, предназначенную для модифицирования структуры таблиц, мы рассмотрим немного позднее. Поэтому прежде чем вы приступите к вводу данных, ознакомьтесь с командной для удаления таблицы.

```
DROP TABLE имя_таблицы;
```

Теперь вы можете приступить к вводу данных в таблицу «Самолеты». Для выполнения этой операции служит команда INSERT. Ее упрощенный формат таков:

```
INSERT INTO имя_таблицы [( имя_атрибута, имя_атрибута, ... )]
  VALUES ( значение_атрибута, значение_атрибута, ... );
```

В начале команды перечисляются атрибуты таблицы. При этом можно указывать их не в том порядке, в котором они были указаны при ее создании. Вы вовсе не обязаны помнить порядок атрибутов в команде CREATE TABLE. Обратите внимание на наличие квадратных скобок. Они указывают, что список атрибутов в команде не является обязательным, но при вводе команды квадратные скобки вводить не нужно. Однако если вы не привели список атрибутов, тогда вы обязаны в предложении VALUES задавать значения атрибутов с учетом того порядка, в котором они следуют в определении таблицы. Конечно, такая форма записи команды является более короткой, но она менее универсальна, т. к. в случае реструктуризации таблицы и изменения порядка столбцов в ее определении или добавления нового столбца (даже без изменения порядка существующих столбцов) вам придется корректировать и команду INSERT в ваших прикладных программах.

Давайте добавим одну строку в таблицу aircrafts. Обратите внимание на одинарные кавычки, в которые заключены значения атрибутов aircraft_code и model. Для атрибутов символьных типов данных одинарные кавычки обязательны, а для числовых типов кавычки использовать не нужно.

```
INSERT INTO aircrafts ( aircraft_code, model, range )
  VALUES ( 'SU9', 'Sukhoi SuperJet-100', 3000 );
```

В ответ мы получим сообщение об успешном добавлении этой строки:

```
INSERT 0 1
```

В этом сообщении числа 0 и 1 имеют конкретный смысл. Второе из них, т. е. 1, означает количество добавленных строк — в данном случае была добавлена всего одна строка. А первое число 0 имеет отношение к внутреннему устройству PostgreSQL, которое в нашем учебном пособии не рассматривается.

Теперь уже можно выполнить выборку данных из таблицы aircrafts. Для выборки информации из таблиц базы данных служит команда SELECT. Ее синтаксис, упрощенный до предела, таков:

```
SELECT имя_атрибута, имя_атрибута, ...
  FROM имя_таблицы;
```

Часто бывает так, что требуется вывести значения из всех столбцов таблицы. В таком случае можно не перечислять имена атрибутов, а просто ввести символ «*». Давайте выберем всю информацию из таблицы aircrafts:

```
SELECT * FROM aircrafts;
```

СУБД ответит таким образом:

```
 aircraft_code |       model          | range
---------------+----------------------+-------
 SU9           | Sukhoi SuperJet-100 |  3000
(1 строка)
```

Давайте добавим еще несколько строк в таблицу aircrafts. Команда INSERT позволяет сделать это за один раз. Вспомните о том, что можно редактировать ранее введенную команду, вызвав ее на экран при помощи клавиши «стрелка вверх». Как и при вводе предыдущих команд, вы можете выбрать один из двух способов ввода: ввести всю команду на одной строке, когда ее текст сворачивается «змейкой», либо вводить

команду построчно, нажимая клавишу Enter после каждого фрагмента команды, занимающего одну строку текста в пособии.

```
INSERT INTO aircrafts ( aircraft_code, model, range )
VALUES ( '773', 'Boeing 777-300', 11100 ),
( '763', 'Boeing 767-300', 7900 ),
( '733', 'Boeing 737-300', 4200 ),
( '320', 'Airbus A320-200', 5700 ),
( '321', 'Airbus A321-200', 5600 ),
( '319', 'Airbus A319-100', 6700 ),
( 'CN1', 'Cessna 208 Caravan', 1200 ),
( 'CR2', 'Bombardier CRJ-200', 2700 );
```

СУБД сообщит об успешном вводе 8 строк в таблицу aircrafts.

```
INSERT 0 8
```

Давайте снова посмотрим, что содержится в таблице «Самолеты».

```
SELECT * FROM aircrafts;
```

Теперь в ней уже 9 строк.

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 SU9           | Sukhoi SuperJet-100 |  3000
 773           | Boeing 777-300      | 11100
 763           | Boeing 767-300      |  7900
 733           | Boeing 737-300      |  4200
 320           | Airbus A320-200     |  5700
 321           | Airbus A321-200     |  5600
 319           | Airbus A319-100     |  6700
 CN1           | Cessna 208 Caravan  |  1200
 CR2           | Bombardier CRJ-200  |  2700
(9 строк)
```

Если сравнить порядок, в котором вы вводили строки в таблицу, с тем порядком, в котором строки выведены из нее по команде SELECT, то можно увидеть совпадение этих порядков. Однако полагаться на такое совпадение нельзя, т. к. порядок может измениться. При выполнении простой выборки из таблицы СУБД не гарантирует никакого конкретного порядка вывода строк. Если же вы хотите каким-то образом упорядочить расположение выводимых строк, то необходимо предпринять дополнительные меры, а именно: использовать предложение ORDER BY команды SELECT.

Давайте упорядочим строки по значению атрибута model, а заодно изменим порядок расположения столбцов в выводе информации. Обратите внимание, что символьные значения при выводе выравниваются по левому краю столбца, а числовые значения — по правому краю.

```
SELECT model, aircraft_code, range
  FROM aircrafts
  ORDER BY model;
```

```
        model         | aircraft_code | range
----------------------+---------------+-------
 Airbus A319-100      | 319           |  6700
 Airbus A320-200      | 320           |  5700
 Airbus A321-200      | 321           |  5600
 Boeing 737-300       | 733           |  4200
 Boeing 767-300       | 763           |  7900
 Boeing 777-300       | 773           | 11100
 Bombardier CRJ-200   | CR2           |  2700
 Cessna 208 Caravan   | CN1           |  1200
 Sukhoi SuperJet-100  | SU9           |  3000
(9 строк)
```

Далеко не всегда требуется выбирать ВСЕ строки из таблицы. Множество выбираемых строк можно ограничить с помощью предложения WHERE команды SELECT. Давайте выберем модели самолетов, у которых максимальная дальность полета находится в пределах от 4 до 6 тысяч км включительно.

```
SELECT model, aircraft_code, range
  FROM aircrafts
  WHERE range >= 4000 AND range <= 6000;
```

Условие выбора строк может быть составным. В данном случае мы скомбинировали два ограничения с помощью логической операции AND (т. е. «И»).

```
 model            | aircraft_code | range
------------------+---------------+-------
 Boeing 737-300   | 733           |  4200
 Airbus A320-200  | 320           |  5700
 Airbus A321-200  | 321           |  5600
(3 строки)
```

Теперь мы ознакомимся с командой UPDATE, предназначенной для обновления данных в таблицах. Ее упрощенный синтаксис таков:

```
UPDATE имя_таблицы
  SET имя_атрибута1 = значение_атрибута1,
      имя_атрибута2 = значение_атрибута2, ...
  WHERE условие;
```

Условие, указываемое в команде, должно ограничить диапазон обновляемых строк. Если это условие не задать, то будут обновлены ВСЕ строки в таблице. Если же вам требуется обновить лишь часть из них, то не забывайте указывать условие отбора строк для обновления.

Давайте предположим, что российские инженеры немного улучшили летные характеристики самолета Sukhoi SuperJet, и теперь дальность его полета стала на 500 км больше.

```
UPDATE aircrafts SET range = 3500
  WHERE aircraft_code = 'SU9';
```

СУБД выведет сообщение, подтверждающее успешное обновление одной строки:

```
UPDATE 1
```

Давайте проверим, что получилось в результате обновления данных.

```
SELECT * FROM aircrafts WHERE aircraft_code = 'SU9';
```

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 SU9           | Sukhoi SuperJet-100 | 3500
(1 строка)
```

Итак, мы добрались до операции удаления строк из таблиц. Для этого используется команда DELETE, которая похожа на команду SELECT:

```
DELETE FROM имя_таблицы WHERE условие;
```

Удалите какую-нибудь одну строку из таблицы «Самолеты» (aircrafts):

```
DELETE FROM aircrafts WHERE aircraft_code = 'CN1';
```

СУБД сообщит об успешном удалении одной строки:

```
DELETE 1
```

Вы можете указать и какое-нибудь более сложное условие. Давайте, например, удалим информацию о самолетах с дальностью полета более 10 000 км, а также с дальностью полета менее 3000 км:

```
DELETE FROM aircrafts WHERE range > 10000 OR range < 3000;
```

При необходимости удаления ВСЕХ строк из таблицы, команда будет совсем простой:

```
DELETE FROM aircrafts;
```

Теперь в таблице «Самолеты» (aircrafts) нет ни одной строки. Для продолжения работы необходимо эти данные восстановить. Можно использовать несколько способов.

1. Ввести заново команды INSERT из текста пособия, которые вы ранее уже вводили.
2. Используя клавиши «стрелка вверх» и «стрелка вниз», найти команды INSERT в списке истории команд и повторно их выполнить.
3. С помощью специальной команды, предусмотренной в утилите psql, сохранить всю историю выполненных вами команд в текстовом файле:

   ```
   \s имя_файла_для_сохранения_истории_команд
   ```

Затем нужно открыть его в текстовом редакторе, найти в файле нужные вам команды INSERT и, копируя команды в буфер обмена, вставить их в командную строку утилиты psql и выполнить.

В нашей учебной базе данных содержится несколько таблиц, связанных между собой. Для таблицы «Самолеты» (aircrafts) ближайшей «родственницей» является таблица «Места» (seats). Она имеет следующую структуру:

| Описание атрибута | Имя атрибута | Тип данных | Тип PostgreSQL | Ограничения |
|---|---|---|---|---|
| Код самолета, IATA | aircraft_code | Символьный | char( 3 ) | NOT NULL |
| Номер места | seat_no | Символьный | varchar( 4 ) | NOT NULL |
| Класс обслуживания | fare_conditions | Символьный | varchar( 10 ) | NOT NULL<br>Значения из списка: Economy, Comfort, Business |

Для атрибута «Код самолета, IATA» был выбран тип char(3), поскольку этот атрибут присутствует и в таблице «Самолеты» (aircrafts). Значения атрибута «Номер места» (seat_no) состоят из числовой части, обозначающей номер ряда кресел в салоне самолета, и латинской буквы, обозначающей позицию в ряду, начиная с буквы «A». Например: 10A, 21D, 17F и т. д. В качестве типа данных для этого атрибута выберем varchar(4). Этот тип позволяет хранить любые символы. В скобках указана предельная длина символьной строки, которую можно ввести в поле такого типа. Значения атрибута «Класс обслуживания» (fare_conditions) могут выбираться из ограниченного списка значений. Проверка на соответствие вводимых значений этому списку будет обеспечиваться с помощью ограничения CHECK. Также выбираем тип данных varchar. Все допустимые значения имеют различные длины, но мы ориентируемся на самое длинное значение.

Значения всех атрибутов каждой строки данной таблицы не должны быть не определенными, поэтому на них накладывается ограничение NOT NULL.

В качестве первичного ключа выбрана комбинация атрибутов «Код самолета, IATA» и «Номер места» — это составной ключ. Таким образом, первичный ключ будет **естественным**. Как уже было сказано выше, это означает, что и в реальной предметной области существуют такие понятия, как код самолета и номер места, и эти понятия используются на практике.

В этой таблице используется **внешний ключ**. Предложение FOREIGN KEY создает ограничение ссылочной целостности. В качестве внешнего ключа служит атрибут «Код самолета» (aircraft_code). Он ссылается на одноименный атрибут в таблице «Самолеты» (aircrafts). Таблица «Места» называется ссылающейся (referencing), а таблица «Самолеты» — ссылочной (referenced). Поскольку номера мест привязаны к модели самолета, то в случае удаления из таблицы «Самолеты» какой-либо строки с конкретным кодом самолета необходимо удалить также и из таблицы «Места» все строки, в которых значение атрибута «Код самолета» такой же. Коротко говоря, если в базе данных нет информации о какой-либо модели самолета, то не может быть и информации о компоновке салона, т. е. о распределении мест по классам обслуживания для этой модели. Поэтому в предложении для определения внешнего ключа появляется важное дополнение: ON DELETE CASCADE. Это означает, что при удалении какой-либо строки из таблицы «Самолеты» удаление строк из таблицы «Места», связанных с этой строкой по внешнему ключу, берет на себя СУБД, избавляя программиста от этой заботы. Подобные действия, которые выполняет сама СУБД, называются каскадным удалением. Таким образом, внешний ключ служит для связи таблиц между собой.

Итак, команда для создания нашей второй таблицы «Места» такова:

```
CREATE TABLE seats
( aircraft_code   char( 3 )     NOT NULL,
  seat_no         varchar( 4 )  NOT NULL,
```

```
  fare_conditions varchar( 10 ) NOT NULL,
  CHECK ( fare_conditions IN ( 'Economy', 'Comfort', 'Business' ) ),
  PRIMARY KEY ( aircraft_code, seat_no ),
  FOREIGN KEY ( aircraft_code )
    REFERENCES aircrafts (aircraft_code )
    ON DELETE CASCADE
);
```

Для того чтобы посмотреть, какая получилась таблица, введите команду

```
\d seats
```

```
Таблица "public.seats"
 Колонка         | Тип                    | Модификаторы
-----------------+------------------------+--------------
 aircraft_code   | character(3)           | NOT NULL
 seat_no         | character varying(4)   | NOT NULL
 fare_conditions | character varying(10) | NOT NULL
Индексы:
  "seats_pkey" PRIMARY KEY, btree (aircraft_code, seat_no)
Ограничения-проверки:
  "seats_fare_conditions_check" CHECK (fare_conditions::text = ANY
 ↪  (ARRAY['Economy'::character varying, 'Comfort'::character varying,
 ↪  'Business'::character varying]::text[]))
Ограничения внешнего ключа:
  "seats_aircraft_code_fkey" FOREIGN KEY (aircraft_code)
     REFERENCES aircrafts(aircraft_code) ON DELETE CASCADE
```

Вы видите, что тип данных char имеет также и полное название — character, а тип данных varchar — character varying. Первичный ключ здесь составной — (aircraft_code, seat_no). Ограничение CHECK, накладываемое на значения атрибута fare_conditions, представлено в более сложной форме, чем это было сделано при создании таблицы. Двойные символы «::» означают операцию приведения типа. Это аналогично такой же операции в других языках программирования. Ключевое слово ARRAY говорит о том, что список допустимых значений представлен в виде массива. Массивы присутствуют в PostgreSQL, и их использование в ряде ситуаций позволяет, например, упростить схему базы данных. Более подробно о них мы будем говорить в главе 4.

Принципиально новым по сравнению с таблицей «Самолеты» является наличие ограничения внешнего ключа. Это ограничение имеет имя seats_aircraft_code_fkey, сгенерированное самой СУБД, поскольку мы не предложили в команде CREATE TABLE никакого своего имени для этого ограничения, хотя, в принципе, имели право это сделать, если бы захотели.

Для просмотра списка всех таблиц, имеющихся в вашей базе данных, выполните команду

```
\d
```

```
Список отношений
  Схема |     Имя    |   Тип   | Владелец
--------+------------+---------+----------
 public | aircrafts | таблица | postgres
 public | seats     | таблица | postgres
(2 строки)
```

В первой колонке выведенной таблицы указана так называемая схема базы данных — public. Мы уже говорили, что схема — это обособленный до некоторой степени раздел базы данных. По умолчанию все объекты создаются в схеме public. В третьей колонке указан тип — «таблица». Кроме таблиц могут быть еще и представления. В последней колонке указано имя пользователя, являющегося владельцем таблицы. Как правило, это пользователь, создавший таблицу.

Давайте сразу же проделаем эксперимент, позволяющий показать работу внешнего ключа. Выполните следующую команду для ввода данных в таблицу «Места»:

```
INSERT INTO seats VALUES ( '123', '1A', 'Business' );
```

СУБД ответит так:

```
ОШИБКА: INSERT или UPDATE в таблице "seats" нарушает ограничение внешнего
 ↪  ключа "seats_aircraft_code_fkey"
ПОДРОБНОСТИ: Ключ (aircraft_code)=(123) отсутствует в таблице
 ↪  "aircrafts".
```

Это совершенно логично: если в таблице «Самолеты», на которую ссылается таблица «Места», нет описания самолета с кодом самолета, равным «123», то добавлять информацию о номерах кресел для такого — несуществующего — самолета не имеет смысла. Так действует поддержка правил ссылочной целостности со стороны СУБД. Программист избавлен от необходимости отслеживать и обеспечивать «вручную» соблюдение этих правил.

Теперь нужно заполнить данными таблицу «Места». Для каждой модели самолетов введите только несколько строк, при этом предусмотрите записи для классов обслуживания «Business» и «Economy». С помощью одной команды INSERT можно ввести сразу несколько строк:

```
INSERT INTO seats VALUES
   ( 'SU9', '1A', 'Business' ),
   ( 'SU9', '1B', 'Business' ),
   ( 'SU9', '10A', 'Economy' ),
   ( 'SU9', '10B', 'Economy' ),
   ( 'SU9', '10F', 'Economy' ),
   ( 'SU9', '20F', 'Economy' );
```

Затем измените значение атрибута aircraft_code на другое, например, «773», и повторите команду INSERT. Так придется поступить со всеми моделями самолетов.

Таблица «Места» заполнена необходимыми данными. Теперь решим еще одну задачу. Предположим, что нам нужно получить информацию о количестве мест в салонах для всех типов самолетов. Имея некоторый опыт в программировании на других языках, нетрудно предположить, что в языке SQL должна присутствовать функция для подсчета количества строк в таблицах. Да, такая функция есть — это count. Конечно, для решения задачи, поставленной выше, в принципе можно воспользоваться такими командами:

```
SELECT count( * ) FROM seats WHERE aircraft_code = 'SU9';
SELECT count( * ) FROM seats WHERE aircraft_code = 'CN1';
... ... ... ...
```

Очевидно, что это нерациональный подход, поскольку придется выполнять отдельные однотипные команды для всех моделей самолетов. Язык SQL позволяет упростить решение такой задачи за счет применения операции группирования строк на основе некоторого критерия. Этим критерием будет являться совпадение значений атрибута «Код самолета» (aircraft_code) у различных строк таблицы «Места» (seats).

В модифицированной команде вместо предложения WHERE будет добавлено предложение GROUP BY, которое отвечает за группировку строк с одинаковыми значениями атрибута aircraft_code. Обратите внимание, что при наличии предложения GROUP BY агрегатная функция count выполняет подсчеты строк *для каждой группы* строк.

```
SELECT aircraft_code, count( * ) FROM seats
  GROUP BY aircraft_code;
```

Конечно, в вашей выборке значения в столбце count будут гораздо меньше.

```
 aircraft_code | count
---------------+-------
 773           |  402
 733           |  130
 CN1           |   12
 CR2           |   50
 319           |  116
 SU9           |   97
 321           |  170
 763           |  222
 320           |  140
(9 строк)
```

Если мы захотим отсортировать выборку по числу мест в самолетах, то нужно будет дополнить команду предложением ORDER BY, которое обеспечит сортировку результирующих строк по значениям второго столбца.

```
SELECT aircraft_code, count( * ) FROM seats
  GROUP BY aircraft_code
  ORDER BY count;
```

```
 aircraft_code | count
---------------+-------
 CN1           |   12
 CR2           |   50
 SU9           |   97
 319           |  116
 733           |  130
 320           |  140
 321           |  170
 763           |  222
 773           |  402
(9 строк)
```

Теперь поставим более сложную задачу: подсчитать количество мест в салонах для всех моделей самолетов, но теперь уже с учетом класса обслуживания (бизнес-класс и экономический класс). В этом случае группировка выполняется уже по двум атрибутам: aircraft_code и fare_conditions. Отсортируем выборку по тем же столбцам, по которым выполняли группировку.

```
SELECT aircraft_code, fare_conditions, count( * )
  FROM seats
  GROUP BY aircraft_code, fare_conditions
  ORDER BY aircraft_code, fare_conditions;
```

```
 aircraft_code | fare_conditions | count
---------------+-----------------+-------
 319           | Business        |   20
 319           | Economy         |   96
 320           | Business        |   20
 320           | Economy         |  120
...
(17 строк)
```

## Контрольные вопросы и задания

1. Попробуйте ввести в таблицу aircrafts строку с таким значением атрибута «Код самолета» (aircraft_code), которое вы уже вводили, например:

   ```
   INSERT INTO aircrafts
     VALUES ( 'SU9', 'Sukhoi SuperJet-100', 3000 );
   ```

   Обратите внимание, что в этой команде мы не привели список атрибутов, что вполне допустимо при задании значений атрибутов в том же порядке, в котором атрибуты следуют в определении таблицы. Но в ваших прикладных программах так поступать все же не следует, поскольку в случае возможной реструктуризации таблицы и изменения порядка следования атрибутов в ней ваши команды INSERT могут перестать работать корректно.

   Вы получите сообщение об ошибке.

   ```
   ОШИБКА: повторяющееся значение ключа нарушает ограничение
    ↪  уникальности "aircrafts_pkey"
   ПОДРОБНОСТИ: Ключ "(aircraft_code)=(SU9)" уже существует.
   ```

   Подумайте, почему оно появилось. Если вы забыли структуру таблицы aircrafts, то можно вывести ее определение на экран с помощью команды

   ```
   \d aircrafts
   ```

2. Предложение ORDER BY команды SELECT позволяет отсортировать данные при выводе. По умолчанию сортировка выполняется по возрастанию значений атрибута, указанного в этом предложении. Но можно упорядочить строки и по убыванию значения атрибута. Для этого нужно после имени атрибута в предложении ORDER BY добавить ключевое слово DESC (это сокращение от слова descendant — убывающий порядок). Самостоятельно напишите команду для выборки всех строк из таблицы aircrafts, чтобы строки были упорядочены по убыванию значения атрибута «Максимальная дальность полета, км» (range).

3. Команда UPDATE позволяет в процессе обновления выполнять арифметические действия над значениями, находящимися в строках таблицы. Представим себе, что двигатели самолета Sukhoi SuperJet стали в два раза экономичнее, вследствие чего дальность полета этого лайнера возросла ровно в два раза. Команда UPDATE позволяет увеличить значение атрибута range в строке, хранящей информацию об этом самолете, даже не выполняя предварительно выборку с целью выяснения текущего значения этого атрибута. При присваивании нового значения атрибуту range можно справа от знака «=» написать не только числовую константу, но и целое выражение. В нашем случае оно будет простым: range = range * 2. Самостоятельно напишите команду UPDATE полностью, при этом не забудьте, что увеличить дальность полета нужно только у одной модели — Sukhoi SuperJet, поэтому необходимо использовать условие WHERE. Затем с помощью команды SELECT проверьте полученный результат.

4. Если в предложении WHERE команды DELETE вы укажете логически и синтаксически корректное условие, но строк, удовлетворяющих этому условия, в таблице не окажется, то в ответ СУБД выведет сообщение

   ```
   DELETE 0
   ```

   Такая ситуация не является ошибкой или сбоем в работе СУБД. Например, если после удаления какой-то строки вы повторно попытаетесь удалить ее же, то получите именно такое сообщение.

   Самостоятельно смоделируйте описанную ситуацию, подобрав условие, которому гарантированно не соответствует ни одна строка в таблице «Самолеты» (aircrafts).

# 4 Типы данных СУБД PostgreSQL

После первоначального знакомства с языком SQL имеет смысл немного упорядочить полученные вами знания. Речь идет о типах данных, применяемых в СУБД PostgreSQL. Вообще, типы данных — это одно из базовых понятий любого языка программирования, и язык SQL в этом плане не является исключением.

PostgreSQL имеет очень разнообразный набор встроенных типов данных, т. е. тех типов, которые СУБД предоставляет в распоряжение пользователя, как говорят, по умолчанию. Мы намеренно употребили здесь термин «встроенные», поскольку пользователь имеет возможность создавать и свои собственные типы данных, которые затем можно включить в систему и использовать их так же, как и встроенные. Такая возможность адаптации системы типов данных к конкретным ситуациям является одной из отличительных черт PostgreSQL.

В этой главе мы расскажем лишь о самых основных типах данных, поскольку в вашем распоряжении всегда имеется полная документация. Настоящее учебное пособие не является ее заменой, оно призвано лишь помочь вам сделать первые шаги в освоении богатого мира типов данных PostgreSQL.

Типы данных объединены в группы, в рамках этих групп они имеют некоторые общие свойства, но также они имеют и различия.

## 4.1 Числовые типы

Группа числовых типов данных включает в себя целый ряд разновидностей: целочисленные типы, числа фиксированной точности, типы данных с плавающей точкой, последовательные типы (serial).

В составе целочисленных типов находятся следующие представители: smallint, integer, bigint. Если атрибут таблицы имеет один из этих типов, то он позволяет хранить только целочисленные данные. При этом перечисленные типы различаются по количеству байтов, выделяемых для хранения данных. В PostgreSQL существуют псевдонимы для этих стандартизированных имен типов, а именно: int2, int4 и int8. Число байтов отражается в имени типа.

При выборе конкретного целочисленного типа принимают во внимание диапазон допустимых значений и затраты памяти. Зачастую тип integer считается оптимальным выбором с точки зрения достижения компромисса между этими показателями.

Числа фиксированной точности представлены двумя типами — numeric и decimal. Однако они являются идентичными по своим возможностям. Поэтому мы будем проводить изложение на примере типа numeric. Для задания значения этого типа используются два базовых понятия: масштаб (scale) и точность (precision). Масштаб показывает число значащих цифр, стоящих справа от десятичной точки (запятой). Точность указывает общее число цифр как до десятичной точки, так и после нее. Например, у числа 12.3456 точность составляет 6 цифр, а масштаб — 4 цифры.

Параметры этого типа данных указываются в скобках: numeric(точность, масштаб). Например, numeric(6, 2).

Его главное достоинство — это обеспечение *точных* результатов при выполнении вычислений, когда это, конечно, возможно в принципе. Это оказывается возможным при выполнении сложения, вычитания и умножения. Числа типа numeric могут хранить очень большое количество цифр: 131072 цифры — до десятичной точки (запятой), 16383 — после точки. Однако нужно учитывать, что такая точность достигается за счет замедления вычислений по сравнению с целочисленными типами и типами с плавающей запятой. При этом для хранения числа затрачивается больше памяти, чем в случае целых чисел.

Данный тип следует выбирать для хранения денежных сумм, а также в других случаях, когда требуется гарантировать точность вычислений.

Представителями типов данных с плавающей точкой являются real и double precision. Они представляют собой реализацию стандарта IEEE «Standard 754 for Binary Floating-Point Arithmetic». Тип данных real может представить числа в диапазоне, как минимум, от 1E-37 до 1E+37 с точностью не меньше 6 десятичных цифр. Тип double precision имеет диапазон значений примерно от 1E-307 до 1E+308 с точностью не меньше 15 десятичных цифр. При попытке записать в такой столбец слишком большое или слишком маленькое значение будет генерироваться ошибка. Если точность вводимого числа выше допустимой, то будет иметь место округление значения. А вот при вводе очень маленьких чисел, которые невозможно представить значениями, отличными от нуля, будет генерироваться ошибка потери значимости, или исчезновения значащих разрядов (an underflow error).

При работе с числами таких типов нужно помнить, что сравнение двух чисел с плавающей точкой на предмет равенства их значений может привести к неожиданным результатам. Например:

```
SELECT 0.1::real * 10 = 1.0::real;

 ?column?
----------
 f
(1 строка)
```

В дополнение к обычным числам эти типы данных поддерживают и специальные значения Infinity (бесконечность), -Infinity (отрицательная бесконечность) и NaN (не число).

PostgreSQL поддерживает также тип данных float, определенный в стандарте SQL. В объявлении типа может использоваться параметр: float(p). Если его значение лежит в диапазоне от 1 до 24, то это будет равносильно использованию типа real, а если же значение лежит в диапазоне от 25 до 53, то это будет равносильно использованию типа double precision. Если же при объявлении типа float параметр не используется, то это также будет равносильно использованию типа double precision.

Последним из числовых типов является тип serial. Однако он фактически реализован не как настоящий тип, а просто как удобная замена целой группы SQL-команд. Тип serial удобен в тех случаях, когда требуется в какой-либо столбец вставлять уникальные целые значения, например, значения суррогатного первичного ключа.

Синтаксис для создания столбца типа serial таков:

```
CREATE TABLE tablename ( colname SERIAL );
```

Эта команда эквивалентна следующей группе команд:

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename
( colname integer NOT NULL
    DEFAULT nextval( 'tablename_colname_seq' )
);
ALTER SEQUENCE tablename_colname_seq
    OWNED BY tablename.colname;
```

Для пояснения вышеприведенных команд нам придется немного забежать вперед. Одним из видов объектов в базе данных являются так называемые последовательности. Это, по сути, генераторы уникальных целых чисел. Для работы с этими последовательностями-генераторами используются специальные функции. Одна из них — это функция nextval(), которая как раз и получает очередное число из последовательности, имя которой указано в качестве параметра функции. В команде CREATE TABLE ключевое слово DEFAULT предписывает, чтобы СУБД использовала в качестве значения по умолчанию то значение, которое формирует функция nextval(). Поэтому если в команде вставки строки в таблицу INSERT INTO не будет передано значение для поля типа serial, то СУБД обратится к услугам этой функции. В том случае, когда в таблице поле типа serial является суррогатным первичным ключом, тогда нет необходимости указывать явное значение для вставки в это поле.

В заключение скажем, что кроме типа serial существуют еще два аналогичных типа: bigserial и smallserial. Им фактически, за кадром, соответствуют типы bigint и smallint. Поэтому при выборе конкретного последовательного типа нужно учитывать предполагаемое число строк в таблице и частоту удаления и вставки строк, поскольку даже для небольшой таблицы может потребоваться большой диапазон, если операции удаления и вставки строк выполняются часто.

## 4.2 Символьные (строковые) типы

Стандартные представители строковых типов — это character varying(n) и character(n), где параметр указывает максимальное число символов в строке, которую можно сохранить в столбце такого типа. При работе с многобайтовыми кодировками символов, например, UTF-8, нужно учитывать, что речь идет именно о символах, а не о байтах. Если сохраняемая строка символов будет короче, чем указано в определении типа, то значение типа character будет дополнено пробелами до требуемой длины, а значение типа character varying будет сохранено так, как есть.

Типы character varying(n) и character(n) имеют псевдонимы varchar(n) и char(n) соответственно. На практике, как правило, используют именно эти краткие псевдонимы.

PostgreSQL дополнительно предлагает еще один символьный тип — text. В столбец этого типа можно ввести сколь угодно большое значение, конечно, в пределах, установленных при компиляции исходных текстов СУБД.

Документация рекомендует использовать типы text и varchar, поскольку такое отличительное свойство типа character, как дополнение значений пробелами, на практике почти не востребовано. В PostgreSQL обычно используется тип text.

Константы символьных типов в SQL-командах заключаются в одинарные кавычки:

```
SELECT 'PostgreSQL';
```

```
 ?column?
------------
 PostgreSQL
(1 строка)
```

В том случае, когда в константе содержится символ одинарной кавычки или обратной косой черты, их необходимо удваивать. Например:

```
SELECT 'PGDAY''17';
```

```
 ?column?
----------
 PGDAY'17
(1 строка)
```

В том случае, когда таких символов в константе много, все выражение становится трудно воспринимать. На помощь может прийти использование удвоенного символа «$». Эти символы выполняют ту же роль, что и одинарные качки, когда в них заключается строковая константа. При использовании символов «$» в качестве ограничителей уже не нужно удваивать никакие символы, содержащиеся в самой константе: ни одинарные кавычки, ни символы обратной косой черты. Например:

```
SELECT $$PGDAY'17$$;
```

```
 ?column?
----------
 PGDAY'17
(1 строка)
```

Возможность использования символов доллара в роли ограничителей символьной константы не является частью стандарта SQL. Это расширение, предлагаемое PostgreSQL. Подробно об этом написано в разделе документации 4.1.2.4 «Строковые константы, заключенные в доллары».

PostgreSQL предлагает еще одно расширение стандарта SQL — строковые константы в стиле языка C. Чтобы иметь возможность их использовать, нужно перед начальной одинарной кавычкой написать символ E. Например, для включения в константу символа новой строки «\n» нужно сделать так:

```
SELECT E'PGDAY\n17';
```

```
 ?column?
----------
 PGDAY   +
 17
(1 строка)
```

При использовании С-стиля необходимо удваивать обратную косую черту, если требуется поместить ее в константу буквально. А для включения в содержимое константы символа обратной кавычки можно либо удвоить ее, либо сделать так:

```
SELECT E'PGDAY\'17';
```

```
 ?column?
----------
 PGDAY'17
(1 строка)
```

Подробно об использовании С-стиля написано в разделе документации 4.1.2.2 «Строковые константы со спецпоследовательностями в стиле С».

## 4.3 Типы «дата/время»

PostgreSQL поддерживает все типы данных, предусмотренные стандартом SQL для даты и времени. Даты обрабатываются в соответствии с григорианским календарем, причем, это делается даже в тех случаях, когда дата относится к тому моменту времени, когда этот календарь в данной стране еще не был принят. Для этих типов данных предусмотрены определенные форматы для ввода значений и для вывода. Причем, эти форматы могут не совпадать. Важно помнить, что при вводе значений их нужно заключать в одинарные кавычки, как и текстовые строки.

Начнем рассмотрение с типа date. Рекомендуемый стандартом ISO 8601 формат ввода дат таков: «yyyy-mm-dd», где символы «y», «m» и «d» обозначают цифру года, месяца и дня соответственно. PostgreSQL позволяет использовать и другие форматы для ввода, например: «Sep 12, 2016», что означает 12 сентября 2016 года. При выводе значений PostgreSQL использует формат по умолчанию, если не предписан другой формат. По умолчанию используется формат, рекомендуемый стандартом ISO 8601: «yyyy-mm-dd».

```
SELECT '2016-09-12'::date;
```

```
   date
------------
 2016-09-12
(1 строка)
```

А в следующем примере используется другой формат ввода, но формат вывода остается тот же самый, поскольку мы его не изменяли:

```
SELECT 'Sep 12, 2016'::date;
```

```
    date
------------
 2016-09-12
(1 строка)
```

Чтобы «сказать» СУБД, что введенное значение является датой, а не простой символьной строкой, мы использовали операцию *приведения типа*. В PostgreSQL она оформляется с использованием двойного символа «двоеточие» и имени того типа, к которому мы приводим данное значение. Важно учесть, что при выполнении приведения типа производится проверка значения на соответствие формату целевого типа и множеству его допустимых значений.

В PostgreSQL предусмотрен целый ряд функций для работы с датами и временем. Например, для получения значения текущей даты служит функция current_date. Ее особенностью является то, что при ее вызове круглые скобки не используются.

```
SELECT current_date;
```

```
    date
------------
 2016-09-21
(1 строка)
```

Если нам требуется вывести дату в другом формате, то для разового преобразования формата можно использовать функцию to_char(), например:

```
SELECT to_char( current_date, 'mm-dd-yyyy' );
```

СУБД выведет:

```
   to_char
------------
 21-09-2016
(1 строка)
```

Обратите внимание, что для демонстрации возможностей СУБД по работе с датами нам даже не потребовалось создавать таблицу, а оказалось достаточно лишь команды SELECT.

Для хранения времени суток служат два типа данных: time и time with time zone. Первый из них хранит только время суток, а второй — дополнительно — еще и часовой пояс. Однако документация на PostgreSQL не рекомендует использовать тип time with time zone, поскольку смещение (offset), соответствующее конкретному часовому поясу, может зависеть от даты перехода на летнее время и обратно, но в этом типе дата отсутствует. При вводе значений времени допустимы различные форматы, например:

```
SELECT '21:15'::time;
```

При выводе СУБД дополнит введенное значение, в котором присутствуют только часы и минуты, секундами.

```
   time
----------
 21:15:00
(1 строка)
```

Чтобы «сказать» СУБД, что введенное значение является значением времени, а не простой символьной строкой, мы опять использовали операцию приведения типа. Предложим СУБД заведомо недопустимое значение времени, например:

```
SELECT '25:15'::time;
```

Получим такое сообщение об ошибке:

```
ОШИБКА: значение поля типа date/time вне диапазона: "25:15"
СТРОКА 1: select '25:15'::time;
                 ^
```

А теперь возьмем значение, которое включает еще и секунды.

```
SELECT '21:15:26'::time;
```

```
  time
----------
 21:15:26
(1 строка)
```

А если число секунд недопустимое, то опять получим сообщение об ошибке.

```
SELECT '21:15:69'::time;
```

```
ОШИБКА: значение поля типа date/time вне диапазона: "21:15:69"
СТРОКА 1: select '21:15:69'::time;
                 ^
```

Время можно вводить не только в 24-часовом формате, но и в 12-часовом, при этом нужно использовать дополнительные суффиксы am и pm. Например:

```
SELECT '10:15:16 am'::time;
```

```
   time
----------
 10:15:16
(1 строка)
```

```
SELECT '10:15:16 pm'::time;
```

```
   time
----------
 22:15:16
(1 строка)
```

Для получения значения текущего времени служит функция current_time. При ее вызове круглые скобки не используются.

```
SELECT current_time;
```

```
       timetz
--------------------
 23:51:57.293522+03
(1 строка)
```

Текущее время выводится с высокой точностью и дополняется числовым значением, соответствующим локальному часовому поясу, который установлен в конфигурационном файле сервера PostgreSQL. В приведенном примере значение часового пояса равно +03, но если ваш компьютер находится в другом часовом поясе, то это значение будет другим, например, для регионов Сибири оно может быть +08.

В результате объединения типов даты и времени получается интегральный тип — временная отметка. Этот тип существует в двух вариантах: с учетом часового пояса — timestamp with time zone, либо без учета часового пояса — timestamp. Для первого варианта существует сокращенное наименование — timestamptz, которое является расширением PostgreSQL. При вводе и выводе значений этого типа данных используются соответствующие форматы ввода и вывода даты и времени. Вот пример с учетом часового пояса:

```
SELECT timestamp with time zone '2016-09-21 22:25:35';
```

```
      timestamptz
------------------------
 2016-09-21 22:25:35+03
(1 строка)
```

Обратите внимание, что хотя мы не указали явно значение часового пояса при вводе данных, при выводе это значение «+03» было добавлено.

А это пример без учета часового пояса:

```
SELECT timestamp '2016-09-21 22:25:35';
```

```
      timestamp
---------------------
 2016-09-21 22:25:35
(1 строка)
```

В рассмотренных примерах мы использовали синтаксис type 'string' для указания конкретного типа простой литеральной константы. Имя типа данных мы указывали не после преобразуемого литерала, а перед ним, например, timestamp '2016-09-21 22:25:35'. Строго говоря, это не является операцией приведения типа, хотя и похоже на нее. Подробно данный вопрос рассмотрен в разделах документации 4.1.2.7 «Константы других типов» и 4.2.9 «Приведения типов».

Для получения значения текущей временно́й отметки (т. е. даты и времени в одном значении) служит функция current_timestamp. Она также вызывается без использования круглых скобок. Приведем пример ее использования.

```
SELECT current_timestamp;
```

```
              now
-------------------------------
 2016-09-27 18:27:37.767739+03
(1 строка)
```

Здесь в выводе присутствует и часовой пояс: «+03».

Оба типа данных — timestamp и timestamptz — занимают один и тот же объем 8 байтов, но значения типа timestamptz хранятся, будучи приведенными к нулевому часовому поясу (UTC), а перед выводом приводятся к часовому поясу пользователя.

На практике при принятии решения о том, какой из этих двух типов — timestamp или timestamptz — использовать, необходимо учитывать, требуется ли значения, хранящиеся в таблице, приводить к местному часовому поясу или не требуется. Например, в расписании авиарейсов указывается местное время как для аэропорта отправления, так и для аэропорта прибытия. Поэтому в таком случае нужно использовать тип timestamp, чтобы это время не приводилось к текущему часовому поясу пользователя, где бы он ни находился.

Из двух этих типов данных чаще используется timestamptz.

Последним типом является interval, который представляет продолжительность отрезка времени между двумя моментами времени. Его формат ввода таков:

```
quantity unit [quantity unit ...] direction
```

Здесь unit означает единицу измерения, а quantity — количество таких единиц. В качестве единиц измерения можно использовать следующие: microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium. Параметр direction может принимать значение ago (т. е. «тому назад») либо быть пустым. Приведем примеры:

```
SELECT '1 year 2 months ago'::interval;
```

```
     interval
------------------
 -1 years -2 mons
(1 строка)
```

Обратите внимание, что параметр ago заставляет СУБД добавить знак «минус» перед всеми полями.

Можно использовать альтернативный формат, предлагаемый стандартом ISO 8601:

```
P [ years-months-days ] [ T hours:minutes:seconds ]
```

Здесь строка должна начинаться с символа «P», а символ «T» разделяет дату и время (все выражение пишется без пробелов). Например:

```
SELECT 'P0001-02-03T04:05:06'::interval;
```

```
           interval
-------------------------------
 1 year 2 mons 3 days 04:05:06
(1 строка)
```

Поскольку интервал — это отрезок времени между двумя временны́ми отметками, то значение этого типа можно получить при вычитании одной временно́й отметки из другой.

```
SELECT ('2016-09-16'::timestamp - '2016-09-01'::timestamp)::interval;
```

```
 interval
----------
 15 days
(1 строка)
```

Как мы уже говорили ранее, в PostgreSQL предусмотрен целый ряд функций для работы с датами и временем. Например, для получения значений текущей даты, текущего времени и текущей временно́й отметки (т. е. даты и времени в одном значении) служат функции current_date, current_time, current_timestamp. Эти функции часто применяются для получения значений по умолчанию при вставке строк в таблицы. Их особенностью является то, что при их вызове круглые скобки не используются. Для получения полной информации обратитесь к документации (раздел 9.9 «Операторы и функции даты/времени»).

Значения временны́х отметок можно усекать с той или иной точностью с помощью функции date_trunc(). Например, для получения текущей временно́й отметки с точностью до одного часа нужно сделать так:

```
SELECT ( date_trunc( 'hour', current_timestamp ) );
```

```
       date_trunc
------------------------
 2016-09-27 22:00:00+03
(1 строка)
```

Из значений временны́х отметок можно с помощью функции extract() извлекать отдельные поля, т. е. год, месяц, день, число часов, минут или секунд и т. д. Например, чтобы извлечь номер месяца, нужно сделать так:

```
SELECT extract( 'mon' FROM timestamp '1999-11-27 12:34:56.123459' );
```

```
 date_part
-----------
        11
(1 строка)
```

Напомним, что выражение timestamp '1999-11-27 12:34:56.123459' не означает операцию приведения типа. Оно присваивает тип данных timestamp литеральной константе.

## 4.4 Логический тип

Логический (boolean) тип может иметь три состояния: «true» и «false», а также неопределенное состояние, которое можно представить значением NULL. Таким образом, тип boolean реализует трехзначную логику.

В качестве состояния «true» могут служить следующие значения: TRUE, 't', 'true', 'y', 'yes', 'on', '1'.

В качестве состояния «false» могут служить следующие значения: FALSE, 'f', 'false', 'n', 'no', 'off', '0'.

В качестве примера создадим таблицу с двумя столбцами и добавим в нее несколько строк:

```
CREATE TABLE databases ( is_open_source boolean, dbms_name text );
```

```
INSERT INTO databases VALUES ( TRUE, 'PostgreSQL' );
INSERT INTO databases VALUES ( FALSE, 'Oracle' );
INSERT INTO databases VALUES ( TRUE, 'MySQL' );
INSERT INTO databases VALUES ( FALSE, 'MS SQL Server' );
```

Теперь выполним выборку всех строк из этой таблицы:

```
SELECT * FROM databases;
```

```
 is_open_source |  dbms_name
----------------+--------------
 t              | PostgreSQL
 f              | Oracle
 t              | MySQL
 f              | MS SQL Server
(4 строки)
```

Выберем только СУБД с открытым исходным кодом:

```
SELECT * FROM databases WHERE is_open_source;
```

```
 is_open_source | dbms_name
----------------+------------
 t              | PostgreSQL
 t              | MySQL
(2 строки)
```

Обратите внимание, что в условии WHERE для проверки логических значений можно не писать выражение WHERE is_open_source = 'yes', а достаточно просто указать имя столбца, содержащего логическое значение: WHERE is_open_source.

## 4.5 Массивы

PostgreSQL позволяет создавать в таблицах такие столбцы, в которых будут содержаться не скалярные значения, а массивы переменной длины. Эти массивы могут быть многомерными и могут содержать значения любого из встроенных типов, а также типов данных, определенных пользователем.

Предположим, что нам необходимо сформировать и сохранить в базе данных в удобной форме графики работы пилотов авиакомпании, т. е. номера дней недели, когда они совершают полеты. Создадим таблицу, в которой эти графики будут храниться в виде единых списков, т. е. в виде одномерных массивов.

```
CREATE TABLE pilots
( pilot_name text,
  schedule integer[]
);
```

```
CREATE TABLE
```

Для указания на то, что это массив, нужно добавить квадратные скобки к наименованию типа данных. При этом задавать число элементов не обязательно.

Давайте добавим в таблицу четыре строки. Массив в команде вставки представлен в виде строкового литерала с указанием типа данных и квадратных скобок, означающих массив. Обратите внимание, что все массивы имеют различное число элементов.

```
INSERT INTO pilots
VALUES ( 'Ivan',  '{ 1, 3, 5, 6, 7 }'::integer[] ),
       ( 'Petr',  '{ 1, 2, 5, 7    }'::integer[] ),
       ( 'Pavel', '{ 2, 5          }'::integer[] ),
       ( 'Boris', '{ 3, 5, 6       }'::integer[] );
```

```
INSERT 0 4
```

Посмотрим, что получилось:

```
SELECT * FROM pilots;
```

```
 pilot_name |  schedule
------------+-------------
 Ivan       | {1,3,5,6,7}
 Petr       | {1,2,5,7}
 Pavel      | {2,5}
 Boris      | {3,5,6}
(4 строки)
```

Предположим, что руководство компании решило, что каждый пилот должен летать 4 раза в неделю. Значит, нам придется обновить значения в таблице. Пилоту по имени Boris добавим один день с помощью операции конкатенации:

```
UPDATE pilots
    SET schedule = schedule || 7
    WHERE pilot_name = 'Boris';
```

```
UPDATE 1
```

Пилоту по имени Pavel добавим один день в конец списка (массива) с помощью функции array_append:

```
UPDATE pilots
    SET schedule = array_append( schedule, 6 )
    WHERE pilot_name = 'Pavel';
```

```
UPDATE 1
```

Ему же добавим один день в начало списка с помощью функции array_ prepend (обратите внимание, что параметры функции поменялись местами):

```
UPDATE pilots
    SET schedule = array_prepend( 1, schedule )
    WHERE pilot_name = 'Pavel';
```

```
UPDATE 1
```

У пилота по имени Ivan имеется лишний день в графике. С помощью функции array_remove удалим из графика пятницу (второй параметр функции указывает *значение* элемента массива, а не индекс):

```
UPDATE pilots
    SET schedule = array_remove( schedule, 5 )
    WHERE pilot_name = 'Ivan';
```

```
UPDATE 1
```

У пилота по имени Petr изменим дни полетов, не изменяя их общего количества. Воспользуемся индексами для работы на уровне отдельных элементов массива. По умолчанию нумерация индексов начинается с единицы, а не с нуля. При необходимости ее можно изменить. К элементам одного и того же массива можно обращаться в предложении SET по отдельности, как будто это разные столбцы.

```
UPDATE pilots
    SET schedule[ 1 ] = 2, schedule[ 2 ] = 3
    WHERE pilot_name = 'Petr';
```

```
UPDATE 1
```

А можно было бы, используя срез (slice) массива, сделать и так:

```
UPDATE pilots
    SET schedule[ 1:2 ] = ARRAY[ 2, 3 ]
    WHERE pilot_name = 'Petr';
```

```
UPDATE 1
```

В вышеприведенной команде запись 1:2 означает индексы первого и последнего элементов диапазона массива. Нотация с использованием ключевого слова ARRAY — это альтернативный способ создания массива (он соответствует стандарту SQL). Таким образом, присваивание новых значений производится сразу целому диапазону элементов массива.

```
SELECT * FROM pilots;
```

```
 pilot_name | schedule
------------+-----------
 Boris      | {3,5,6,7}
 Pavel      | {1,2,5,6}
 Ivan       | {1,3,6,7}
 Petr       | {2,3,5,7}
(4 строки)
```

Теперь продемонстрируем основные операции, которые можно применять к массивам, выполняя выборки из таблиц.

Получим список пилотов, которые летают каждую среду:

```
SELECT * FROM pilots
    WHERE array_position( schedule, 3 ) IS NOT NULL;
```

```
 pilot_name | schedule
------------+-----------
 Boris      | {3,5,6,7}
 Ivan       | {1,3,6,7}
 Petr       | {2,3,5,7}
(3 строки)
```

Функция array_position возвращает индекс первого вхождения элемента с указанным значением в массив. Если же такого элемента нет, она возвратит NULL.

Выберем пилотов, летающих по понедельникам и воскресеньям:

```
SELECT *
    FROM pilots
    WHERE schedule @> '{ 1, 7 }'::integer[];
```

```
 pilot_name | schedule
------------+-----------
 Ivan       | {1,3,6,7}
(1 строка)
```

Оператор @> означает проверку того факта, что в левом массиве содержатся все элементы правого массива. Конечно, при этом в левом массиве могут находиться и другие элементы, что мы и видим в графике этого пилота.

Еще аналогичный вопрос: кто летает по вторникам и/или по пятницам? Для получения ответа воспользуемся оператором &&, который проверяет наличие общих элементов у массивов, т. е. пересекаются ли их множества значений. В нашем примере число общих элементов, если они есть, может быть равно одному или двум. Здесь мы также использовали нотацию с ключевым словом ARRAY, а не '{ 2, 5 }'::integer[]. Вы можете применять ту, которая принята в рамках выполнения вашего проекта.

```
SELECT *
    FROM pilots
    WHERE schedule && ARRAY[ 2, 5 ];
```

```
 pilot_name | schedule
------------+-----------
 Boris      | {3,5,6,7}
 Pavel      | {1,2,5,6}
 Petr       | {2,3,5,7}
(3 строки)
```

Сформулируем вопрос в форме отрицания: кто не летает ни во вторник, ни в пятницу? Для получения ответа добавим в предыдущую SQL-команду отрицание NOT:

```
SELECT *
    FROM pilots
    WHERE NOT ( schedule && ARRAY[ 2, 5 ] );
```

```
 pilot_name | schedule
------------+-----------
 Ivan       | {1,3,6,7}
(1 строка)
```

Иногда требуется развернуть массив в виде столбца таблицы. В таком случае поможет функция unnest:

```
SELECT unnest( schedule ) AS days_of_week
    FROM pilots
    WHERE pilot_name = 'Ivan';
```

```
 days_of_week
--------------
 1
 3
 6
 7
(4 строки)
```

Подробно использование массивов рассмотрено в документации в разделах 8.15 «Массивы» и 9.18 «Функции и операторы для работы с массивами».

## 4.6 Типы JSON

Типы JSON предназначены для сохранения в столбцах таблиц базы данных таких значений, которые представлены в формате JSON (JavaScript Object Notation). Существует два типа: json и jsonb. Основное различие между ними заключается в быстродействии. Если столбец имеет тип json, тогда сохранение значений происходит быстрее, потому что они записываются в том виде, в котором были введены. Но при последующем использовании этих значений в качестве операндов или параметров функций будет каждый раз выполняться их разбор, что замедляет работу. При использовании типа jsonb разбор производится однократно, при записи значения в таблицу. Это несколько замедляет операции вставки строк, в которых содержатся значения данного типа. Но все последующие обращения к сохраненным значениям выполняются быстрее, т. к. выполнять их разбор уже не требуется.

Есть еще ряд отличий, в частности, тип json сохраняет порядок следования ключей в объектах и повторяющиеся значения ключей, а тип jsonb этого не делает. Рекомендуется в приложениях использовать тип jsonb, если только нет каких-то особых аргументов в пользу выбора типа json.

Для иллюстрации использования типов JSON обратимся к тематике авиаперевозок. Предположим, что руководство авиакомпании всемерно поддерживает стремление пилотов улучшать свое здоровье, повышать уровень культуры и расширять кругозор. Поэтому разработчики базы данных авиакомпании получили задание создать специальную таблицу, в которую будут заноситься сведения о тех видах спорта, которыми занимается пилот, будет отмечаться наличие у него домашней библиотеки, а также фиксироваться количество стран, которые он посетил в ходе туристических поездок.

```
CREATE TABLE pilot_hobbies
( pilot_name text,
  hobbies jsonb
);
```

```
CREATE TABLE
```

```
INSERT INTO pilot_hobbies
    VALUES ( 'Ivan',
             '{ "sports": [ "футбол", "плавание" ],
                "home_lib": true, "trips": 3
              }'::jsonb ),
           ( 'Petr',
             '{ "sports": [ "теннис", "плавание" ],
                "home_lib": true, "trips": 2
              }'::jsonb ),
           ( 'Pavel',
             '{ "sports": [ "плавание" ],
                "home_lib": false, "trips": 4
              }'::jsonb ),
           ( 'Boris',
             '{ "sports": [ "футбол", "плавание", "теннис" ],
                "home_lib": true, "trips": 0
              }'::jsonb );
```

```
INSERT 0 4
```

```
SELECT * FROM pilot_hobbies;
```

```
 pilot_name |                    i   hobbies
------------+-----------------------------------------------------
 Ivan       | {"trips": 3, "sports": ["футбол", "плавание"],
              "home_lib": true}
 Petr       | {"trips": 2, "sports": ["теннис", "плавание"],
              "home_lib": true}
 Pavel      | {"trips": 4, "sports": ["плавание"],
              "home_lib": false}
 Boris      | {"trips": 0, "sports": ["футбол", "плавание",
              "теннис"], "home_lib": true}
(4 строки)
```

Как видно, при выводе строк из таблицы порядок ключей в JSON-объектах не был сохранен.

Предположим, что нужно сформировать футбольную сборную команду нашей авиакомпании для участия в турнире. Мы можем выбрать всех футболистов таким способом:

```
SELECT * FROM pilot_hobbies
    WHERE hobbies @> '{ "sports": [ "футбол" ] }'::jsonb;
```

```
 pilot_name |                    i   hobbies
------------+-----------------------------------------------------
 Ivan       | {"trips": 3, "sports": ["футбол", "плавание"],
              "home_lib": true}
 Boris      | {"trips": 0, "sports": ["футбол", "плавание",
              "теннис"], "home_lib": true}
(2 строки)
```

Можно было эту задачу решить и таким способом:

```
SELECT pilot_name, hobbies->'sports' AS sports
    FROM pilot_hobbies
    WHERE hobbies->'sports' @> '[ "футбол" ]'::jsonb;
```

```
 pilot_name |              sports
------------+----------------------------------
 Ivan       | ["футбол", "плавание"]
 Boris      | ["футбол", "плавание", "теннис"]
(2 строки)
```

В этом решении мы выводим только информацию о спортивных предпочтениях пилотов. Внимательно посмотрите, как используются одинарные и двойные кавычки. Операция «->» служит для обращения к конкретному ключу JSON-объекта.

При создании столбца с типом данных json или jsonb не требуется задавать структуру объектов, т. е. конкретные имена ключей. Поэтому в принципе возможна ситуация, когда в разных строках в JSON-объектах будут использоваться различные наборы ключей. В нашем примере структуры JSON-объектов во всех строках совпадают. А если бы они не совпадали, то как можно было бы проверить наличие ключа? Продемонстрируем это.

Ключа «sport» в наших объектах нет. Что покажет вызов функции count?

```
SELECT count( * )
    FROM pilot_hobbies
    WHERE hobbies ? 'sport';
```

```
 count
-------
 0
(1 строка)
```

А вот ключ «sports» присутствует. Выполним ту же проверку:

```
SELECT count( * )
    FROM pilot_hobbies
    WHERE hobbies ? 'sports';
```

Да, так и есть. Такие записи найдены.

```
 count
-------
 4
(1 строка)
```

А как выполнять обновление JSON-объектов в строках таблицы? Предположим, что пилот по имени Boris решил посвятить себя только хоккею. Тогда в базе данных мы выполним такую операцию:

```
UPDATE pilot_hobbies
    SET hobbies = hobbies || '{ "sports": [ "хоккей" ] }'
    WHERE pilot_name = 'Boris';
```

```
UPDATE 1
```

Проверим, что получилось:

```
SELECT pilot_name, hobbies
    FROM pilot_hobbies
    WHERE pilot_name = 'Boris';
```

```
 pilot_name |                        hobbies
------------+-----------------------------------------------------
 Boris      | {"trips": 0, "sports": ["хоккей"], "home_lib": true}
(1 строка)
```

Если впоследствии Boris захочет возобновить занятия футболом, то с помощью функции jsonb_set можно будет обновить сведения о нем в таблице:

```
UPDATE pilot_hobbies
    SET hobbies = jsonb_set( hobbies, '{ sports, 1 }', '"футбол"' )
    WHERE pilot_name = 'Boris';
```

```
UPDATE 1
```

Второй параметр функции указывает путь в пределах JSON-объекта, куда нужно добавить новое значение. В данном случае этот путь состоит из имени ключа (sports) и номера добавляемого элемента в массиве видов спорта (номер 1). Нумерация элементов начинается с нуля. Третий параметр имеет тип jsonb, поэтому его литерал заключается в одинарные кавычки, а само добавляемое значение берется в двойные кавычки. В результате получается — '"футбол"'.

Проверим успешность выполнения этой операции:

```
SELECT pilot_name, hobbies
    FROM pilot_hobbies
    WHERE pilot_name = 'Boris';
```

```
 pilot_name |                    i   hobbies
------------+-----------------------------------------------------
 Boris      | {"trips": 0, "sports": ["хоккей", "футбол"],
              "home_lib": true}
(1 строка)
```

Подробно использование типов JSON рассмотрено в документации в разделах 8.14 «Типы JSON» и 9.15 «Функции и операторы JSON».

## Контрольные вопросы и задания

1. Создайте таблицу, содержащую атрибут типа numeric(precision, scale). Пусть это будет таблица, содержащая результаты каких-то измерений. Команда может быть, например, такой:

   ```
   CREATE TABLE test_numeric
   ( measurement numeric(5, 2),
     description text
   );
   ```

   Попробуйте с помощью команды INSERT продемонстрировать округление вводимого числа до той точности, которая задана при создании таблицы.

   Подумайте, какая из следующих команд вызовет ошибку и почему? Проверьте свои предположения, выполнив эти команды.

```
INSERT INTO test_numeric
    VALUES ( 999.9999, 'Какое-то измерение ' );
INSERT INTO test_numeric
    VALUES ( 999.9009, 'Еще одно измерение' )
INSERT INTO test_numeric
    VALUES ( 999.1111, 'И еще измерение' );
INSERT INTO test_numeric
    VALUES ( 998.9999, 'И еще одно' );
```

Продемонстрируйте генерирование ошибки при попытке ввода числа, количество цифр в котором слева от запятой (десятичной точки) превышает допустимое.

2. Предположим, что возникла необходимость хранить в одном столбце таблицы данные, представленные с различной точностью. Это могут быть, например, результаты физических измерений разнородных показателей или различные медицинские показатели здоровья пациентов (результаты анализов). В таком случае можно использовать тип numeric без указания масштаба и точности. Команда для создания таблицы может быть, например, такой:

```
CREATE TABLE test_numeric
( measurement numeric,
  description text
);
```

Если у вас в базе данных уже есть таблица с таким же именем, то можно предварительно ее удалить с помощью команды

```
DROP TABLE test_numeric;
```

Вставьте в таблицу несколько строк:

```
INSERT INTO test_numeric
    VALUES ( 1234567890.0987654321,
            'Масштаб 20 знаков, точность 10 знаков' );

INSERT INTO test_numeric
    VALUES ( 1.5, 'Масштаб 2 знака, точность 1 знак' );

INSERT INTO test_numeric
    VALUES ( 0.12345678901234567890,
            'Масштаб 21 знак, точность 20 знаков' );

INSERT INTO test_numeric
    VALUES ( 1234567890,
            'Масштаб 10 знаков, точность 0 знаков (целое число)' );
```

Теперь сделайте выборку из таблицы и посмотрите, что все эти разнообразные значения сохранены именно в том виде, как вы их вводили.

3. Тип данных numeric поддерживает специальное значение NaN, которое означает «не число» (not a number). В документации утверждается, что значение NaN считается равным другому значению NaN, а также что значение NaN считается бóльшим любого другого «нормального» значения, т. е. не-NaN. Проверьте эти утверждения с помощью SQL команды SELECT.

В качестве примера приведем команду:

```
SELECT 'NaN'::numeric > 10000;
```

```
 ?column?
----------
 t
(1 строка)
```

4. При работе с числами типов real и double precision нужно помнить, что сравнение двух чисел с плавающей точкой на предмет равенства их значений может привести к неожиданным результатам. Например, сравним два очень маленьких числа (они представлены в экспоненциальной форме записи):

```
SELECT '5e-324'::double precision > '4e-324'::double precision;
```

```
 ?column?
----------
 f
(1 строка)
```

Чтобы понять, почему так получается, выполните еще два запроса.

```
SELECT '5e-324'::double precision;
```

```
 float8
-----------------------
 4.94065645841247e-324
(1 строка)
```

```
SELECT '4e-324'::double precision;
```

```
 float8
-----------------------
 4.94065645841247e-324
(1 строка)
```

Самостоятельно проведите аналогичные эксперименты с очень большими числами, находящимися на границе допустимого диапазона для чисел типов real и double precision.

5. Типы данных real и double precision поддерживают специальные значения Infinity (бесконечность) и -Infinity (отрицательная бесконечность). Проверьте с помощью SQL-команды SELECT ожидаемые свойства этих значений. Например, сравните Infinity с наибольшим значением, которое допускается для типа double precision (можно использовать сокращенное написание Inf):

```
SELECT 'Inf'::double precision > 1E+308;
```

```
 ?column?
----------
 t
(1 строка)
```

Выполните аналогичный запрос для наименьшего возможного значения типа double precision.

6. Типы данных real и double precision поддерживают специальное значение NaN, которое означает «не число» (not a number).

   В математике существует такое понятие, как *неопределенность*. В качестве одного из ее вариантов служит результат операции умножения нуля на бесконечность. Посмотрите, что выдаст в результате PostgreSQL:

   ```
   SELECT 0.0 * 'Inf'::real;
   ```

   ```
    ?column?
   ----------
    NaN
   (1 строка)
   ```

   В документации утверждается, что значение NaN считается равным другому значению NaN, а также что значение NaN считается бóльшим любого другого «нормального» значения, т. е. не-NaN. Проверьте эти утверждения с помощью SQL-команды SELECT.

   Например, сравните значения NaN и Infinity.

   ```
   select 'NaN'::real > 'Inf'::real;
   ```

   ```
    ?column?
   ----------
    t
   (1 строка)
   ```

7. Тип serial может применяться для столбцов, содержащих числовые значения, которые должны быть уникальными в пределах таблицы, например, идентификаторы каких-то объектов. В качестве иллюстрация применения типа serial предложим таблицу, содержащую наименования улиц и площадей:

   ```
   CREATE TABLE test_serial ( id serial, name text );
   ```

   Введите несколько строк. Обратите внимание, что значение для столбца id указывать не обязательно (и даже не нужно). Но поскольку мы задаем значения не для всех столбцов, имеющихся в таблице, мы должны указать в команде INSERT не только список значений, но и список столбцов. Конечно, в данном простом случае эти списки состоят лишь из одного элемента.

   ```
   INSERT INTO test_serial ( name ) VALUES ( 'Вишневая' );
   INSERT INTO test_serial ( name ) VALUES ( 'Грушевая' );
   INSERT INTO test_serial ( name ) VALUES ( 'Зеленая' );
   ```

   Сделайте выборку данных из таблицы, вы увидите, что значения столбца id имеют последовательные значения, начиная с 1.

   Давайте проведем эксперимент со столбцом id. Выполните команду INSERT, в которой укажите явное значение столбца id:

   ```
   INSERT INTO test_serial ( id, name ) VALUES ( 10, 'Прохладная' );
   ```

   А теперь добавьте еще одну строку, но уже не указывая явно значение для столбца id (как мы поступали в предыдущих командах):

   ```
   INSERT INTO test_serial ( name ) VALUES ( 'Луговая' );
   ```

Вы увидите, что явное задание значения для столбца id не влияет на автоматическое генерирование значений этого столбца.

8. Немного усложним определение таблицы из предыдущего задания. Пусть теперь столбец id будет первичным ключом этой таблицы.

```
CREATE TABLE test_serial ( id serial PRIMARY KEY, name text );
```

Теперь выполните следующие команды для добавления строк в таблицу и удаления одной строки из нее. Для пошагового управления этим процессом выполняйте выборку данных из таблицы с помощью команды SELECT после каждой команды вставки или удаления.

```
INSERT INTO test_serial ( name ) VALUES ( 'Вишневая' );
```

Явно зададим значение столбца id:

```
INSERT INTO test_serial ( id, name ) VALUES ( 2, 'Прохладная' );
```

При выполнении этой команды СУБД выдаст сообщение об ошибке. Почему?

```
INSERT INTO test_serial ( name ) VALUES ( 'Грушевая' );
```

Повторим эту же команду. Теперь все в порядке. Почему?

```
INSERT INTO test_serial ( name ) VALUES ( 'Грушевая' );
```

Добавим еще одну строку.

```
INSERT INTO test_serial ( name ) VALUES ( 'Зеленая' );
```

А теперь удалим ее же.

```
DELETE FROM test_serial WHERE id = 4;
```

Добавим последнюю строку.

```
INSERT INTO test_serial ( name ) VALUES ( 'Луговая' );
```

Теперь сделаем выборку.

```
SELECT * FROM test_serial;
```

Вы увидите, что в нумерации образовалась «дыра». Это из-за того, что при формировании нового значения из последовательности поиск максимального значения, уже имеющегося в столбце, не выполняется.

```
 id |   name
----+------------
 1  | Вишневая
 2  | Прохладная
 3  | Грушевая
 5  | Луговая
(4 строки)
```

9. Какой календарь используется в PostgreSQL для работы с датами: юлианский или григорианский?

10. Каждый тип данных из группы «дата/время» имеет ограничение на минимальное и максимальное допустимое значение. Найдите в документации в разделе 8.5 «Типы даты/времени» эти значения и подумайте, почему они таковы.

11. Типы timestamp, time и interval позволяют задать точность ввода и вывода значений. Точность предписывает количество десятичных цифр в поле секунд. Проиллюстрируем эту возможность на примере типа time, выполнив три запроса: в первом запросе вообще не используем параметр точности, во втором назначим его равным 0, в третьем запросе сделаем его равным 3:

```
SELECT current_time;
```

```
      timetz
--------------------
 19:46:14.584641+03
(1 строка)
```

```
SELECT current_time::time( 0 );
```

```
   time
----------
 19:39:45
(1 строка)
```

```
SELECT current_time::time( 3 );
```

```
     time
--------------
 19:39:54.085
(1 строка)
```

Выполните подобные команды для типов timestamp и interval.

Тип date такой возможности — задавать точность — не имеет. Как вы думаете, почему?

12.* Формат ввода и вывода даты можно изменить с помощью параметра datestyle. Значение этого параметра состоит из двух компонентов: первый управляет форматом вывода даты, а второй регулирует порядок следования составных частей даты (год, месяц, день) при вводе и выводе. Текущее значение этого параметра можно узнать с помощью команды SHOW:

```
SHOW datestyle;
```

По умолчанию он имеет такое значение:

```
 DateStyle
-----------
 ISO, DMY
(1 строка)
```

Продемонстрируем влияние этого параметра на работу с типами date и timestamp. Для экспериментов возьмем дату, в которой число (день) превышает 12, чтобы нельзя было день перепутать с номером месяца. Пусть это будет, например, 18 мая 2016 г.

```
SELECT '18-05-2016'::date;
```

Хотя порядок следования составных частей даты задан в виде «DMY», т. е. «день, месяц, год», но при выводе он изменяется на «год, месяц, день».

```
    date
------------
 2016-05-18
(1 строка)
```

Попробуем ввести дату в порядке «месяц, день, год»:

```
SELECT '05-18-2016'::date;
```

В ответ получим сообщение об ошибке. Если бы мы выбрали дату, в которой число (день) было бы не больше 12, например, 9, то сообщение об ошибке не было бы сформировано, т. е. мы с такой датой не смогли бы проиллюстрировать влияние значения «DMY» параметра datestyle. Но главное, что в таком случае мы бы просто не заметили допущенной ошибки.

А вот использовать порядок «год, месяц, день» при вводе можно несмотря на то, что параметр datestyle предписывает «день, месяц, год». Порядок «год, месяц, день» является универсальным, его можно использовать всегда, независимо от настроек параметра datestyle.

```
SELECT '2016-05-18'::date;
```

```
    date
------------
 2016-05-18
(1 строка)
```

Продолжим экспериментирование с параметром datestyle. Давайте изменим его значение. Сделать это можно многими способами, но мы упомянем лишь некоторые:

- изменив его значение в конфигурационном файле postgresql.conf, который в нашей инсталляции PostgreSQL, описанной в главе 2, находится в каталоге /usr/local/pgsql/data;
- назначив переменную системного окружения PGDATESTYLE;
- воспользовавшись командой SET.

Сейчас выберем третий способ, а первые два рассмотрим при выполнении других заданий. Поскольку параметр datestyle состоит фактически из двух частей, которые можно задавать не только обе сразу, но и по отдельности, изменим только порядок следования составных частей даты, не изменяя формат вывода с «ISO» на какой-либо другой.

```
SET datestyle TO 'MDY';
```

Повторим одну из команд, выполненных ранее. Теперь она должна вызвать ошибку. Почему?

```
SELECT '18-05-2016'::date;
```

А такая команда, наоборот, теперь будет успешно выполнена:

```
SELECT '05-18-2016'::date;
```

Теперь приведите настройку параметра datestyle в исходное состояние:

```
SET datestyle TO DEFAULT;
```

Самостоятельно выполните команды SELECT, приведенные выше, но замените в них тип date на тип timestamp. Вы увидите, что дата в рамках типа timestamp обрабатывается аналогично типу date.

Сейчас изменим сразу обе части параметра datestyle:

```
SET datestyle TO 'Postgres, DMY';
```

Проверьте полученный результат с помощью команды SHOW.

Самостоятельно выполните команды SELECT, приведенные выше, как для значения типа date, так и для значения типа timestamp. Обратите внимание, что если выбран формат «Postgres», то порядок следования составных частей даты (день, месяц, год), заданный в параметре datestyle, используется не только при вводе значений, но и при выводе. Напомним, что вводом мы считаем команду SELECT, а выводом — результат ее выполнения, выведенный на экран.

В документации (см. раздел 8.5.2 «Вывод даты/времени») сказано, что формат вывода даты может принимать значения «ISO», «Postgres», «SQL» и «German». Первые два варианта мы уже рассмотрели. Самостоятельно поэкспериментируйте с двумя оставшимися по той же схеме, по которой вы уже действовали ранее при выполнении этого задания. Можно воспользоваться и стандартными функциями current_date и current_timestamp.

13. Установить новое значение параметра datestyle можно с помощью создания переменной системного окружения PGDATESTYLE. Назначить эту переменную можно в конфигурационных файлах операционной системы. Но если нам нужно сделать это только на время текущего сеанса работы клиентской программы, например, утилиты psql, то можно ввести значение этой переменной непосредственно в командной строке:

```
PGDATESTYLE="Postgres" psql -d test -U имя_пользователя
```

Проделайте эти действия, а затем уже из командной строки утилиты psql проверьте текущее значение параметра datestyle с помощью команды SHOW.

14. Назначить значение параметра datestyle можно в конфигурационном файле postgresql.conf, который находится в каталоге /usr/local/pgsql/data. Предварительно сохраните текущую (корректно работающую) версию этого файла, а затем измените в нем значение параметра datestyle, например, на «Postgres, YMD». Перезапустите сервер PostgreSQL, чтобы изменения вступили в силу. Для проверки полученного результата выполните несколько команд SELECT, например:

```
SELECT '05-18-2016'::timestamp;
SELECT current_timestamp;
```

15. В документации в разделе 9.8 «Функции форматирования данных» представлены описания множества полезных функций, позволяющих преобразовать в строку данные других типов, например, timestamp. Одна из таких функций — to_char().

    Приведем несколько команд, иллюстрирующих использование этой функции. Ее первым параметром является форматируемое значение, а вторым — шаблон, описывающий формат, в котором это значение будет представлено при вводе или выводе. Сначала попробуйте разобраться, не обращаясь к документации, в том, что означает второй параметр этой функции в каждой из приведенных команд, а затем проверьте свои предположения по документации.

    ```
    SELECT to_char( current_timestamp, 'mi:ss' );

     to_char
    ---------
     47:43
    (1 строка)

    SELECT to_char( current_timestamp, 'dd' );

     to_char
    ---------
     12
    (1 строка)

    SELECT to_char( current_timestamp, 'yyyy-mm-dd' );

     to_char
    ------------
     2017-03-12
    (1 строка)
    ```

    Поэкспериментируйте с этой функцией, извлекая из значения типа timestamp различные поля и располагая их в нужном вам порядке.

16. При выполнении приведения типа данных производится проверка значения на допустимость. Попробуйте ввести недопустимое значение даты, например, 29 февраля в невисокосном году.

    ```
    SELECT 'Feb 29, 2015'::date;
    ```

    Получите сообщение об ошибке.

17. При выполнении приведения типа данных производится проверка значения на допустимость. Попробуйте ввести недопустимое значение времени, например, с нарушением формата.

    ```
    SELECT '21:15:16:22'::time;

    ОШИБКА: неверный синтаксис для типа time: "21:15:16:22"
    СТРОКА 1: select '21:15:16:22'::time;
                     ^
    ```

18. Как вы думаете, значение какого типа будет получено при вычитании одной даты из другой? Например:

```
SELECT ( '2016-09-16'::date - '2016-09-01'::date );
```

Сначала попробуйте получить ответ, рассуждая логически, а затем проверьте на практике в утилите psql.

19. С типами даты и времени можно выполнять различные арифметические операции. Как правило, их применение является интуитивно понятным. Выполните следующую команду и проанализируйте результат.

```
SELECT ( '20:34:35'::time - '19:44:45'::time );
```

А теперь попробуйте предположить, какой результат будет получен, если в этой команде знак «минус» заменить на знак «плюс»? Проверьте ваши предположения с помощью утилиты psql. Подробное описание всех допустимых арифметических операций с датами и временем приведено в документации в разделе 9.9 «Операторы и функции даты/времени».

20. Значение типа interval можно получить при вычитании одной временно́й отметки из другой, например:

```
SELECT ( current_timestamp - '2016-01-01'::timestamp )
AS new_date;
```

```
       new_date
------------------------
 278 days 00:10:33.33236
(1 строка)
```

А что получится, если прибавить интервал к временно́й отметке? Сначала попробуйте дать ответ, не прибегая к помощи утилиты psql, а затем проверьте свой ответ с помощью этой утилиты. Например, прибавим интервал длительностью в 1 месяц к текущей к временно́й отметке:

```
SELECT ( current_timestamp + '1 mon'::interval ) AS new_date;
```

В этой команде с помощью ключевого слова AS мы назначили псевдоним для того столбца, который будет выведен в результате. Выполните эту же команду, убрав псевдоним, и найдите отличия.

21. Можно с высокой степенью уверенности предположить, что при прибавлении интервалов к датам и временны*м* отметкам PostgreSQL учитывает тот факт, что различные месяцы имеют различное число дней. Но как это реализуется на практике? Например, что получится при прибавлении интервала в 1 месяц к последнему дню января и к последнему дню февраля? Сначала сделайте обоснованные предположения о результатах следующих двух команд, а затем проверьте предположения на практике и проанализируйте полученные результаты:

```
SELECT ( '2016-01-31'::date + '1 mon'::interval ) AS new_date;
SELECT ( '2016-02-29'::date + '1 mon'::interval ) AS new_date;
```

22. Форматом ввода и вывода интервалов управляет параметр intervalstyle. Его можно изменить с помощью способов, аналогичных тем, что были описаны выше для параметра datestyle. Самостоятельно поэкспериментируйте с различными значениями параметра intervalstyle аналогично тому, как вы это делали с параметром datestyle. Используйте раздел 8.5 «Типы даты/времени» в документации.

Напомним, что вернуть исходное значение этого параметра в psql можно с помощью команды

```
SET intervalstyle TO DEFAULT;
```

23. Выполните следующие две команды и объясните различия в выведенных результатах:

```
SELECT ( '2016-09-16'::date - '2015-09-01'::date );
SELECT ( '2016-09-16'::timestamp - '2015-09-01'::timestamp );
```

24. Выполните следующие две команды и объясните различия в выведенных результатах:

```
SELECT ( '20:34:35'::time - 1 );
SELECT ( '2016-09-16'::date - 1 );
```

Почему при выполнении первой команды возникает ошибка? Как можно модифицировать эту команду, чтобы ошибка исчезла?

Для получения полной информации обратитесь к разделу «9.9. Операторы и функции даты/времени» в документации.

25. Значения временны*х* отметок можно усекать с той или иной точностью с помощью функции date_trunc(). Например, с помощью следующей команды можно «отрезать» дробную часть секунды:

```
SELECT ( date_trunc( 'sec',
                     timestamp '1999-11-27 12:34:56.987654' ) );
```

```
     date_trunc
---------------------
 1999-11-27 12:34:56
(1 строка)
```

Напомним, что в данной команде используется операция приведения типа.

Выполните эту команду, последовательно указывая в качестве первого параметра значения microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium (которые обозначают соответственно микросекунды, миллисекунды, секунды, минуты, часы, дни, недели, месяцы, годы, десятилетия, века и тысячелетия). Допустимы сокращения sec, min, mon, dec, cent, mil. Обратите внимание, что результирующее значение получается не путем округления исходного значения, а именно путем отбрасывания более мелких единиц. При этом поля времени (часы, минуты и секунды) заменяются нулями, а поля даты (годы, месяцы и дни) — заменяются цифрами «01». Однако при использовании параметра week картина получается более интересная.

26. Функция date_trunc() может работать не только с данными типа timestamp, но также и данными типа interval. Самостоятельно ознакомьтесь с этими возможностями по документации (см. раздел 9.9 «Операторы и функции даты/времени»).

27. Весьма полезна функция extract(). С ее помощью можно извлечь значение отдельного поля из временной отметки timestamp. Наименование поля задается в первом параметре. Эти наименования такие же, что и для функции date_trunc(). Выполните следующую команду

```
SELECT extract(
  'microsecond' from  timestamp '1999-11-27 12:34:56.123459'
);
```

Она выводит не просто значение поля микросекунд, т. е. 123459, а дополнительно преобразует число секунд в микросекунды и добавляет значение поля микросекунд.

```
 date_part
-----------
 56123459
(1 строка)
```

Выполните эту команду, последовательно указывая в качестве первого параметра значения microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium. Можно использовать сокращения этих наименований, которые приведены в предыдущем задании. Обратите внимание, что в ряде случаев выводится не просто конкретное поле (фрагмент) из временной отметки, а некоторый продукт переработки этого поля. Например, если в качестве первого параметра функции extract() в вышеприведенной команде указать cent (век), то мы получим в ответ не 19 (что и было бы буквальным значением поля «век»), а 20, поскольку 1999 год принадлежит двадцатому веку.

28. Функция extract() может работать не только с данными типа timestamp, но также и данными типа interval. Самостоятельно ознакомьтесь с этими возможностями по документации (см. раздел 9.9 «Операторы и функции даты/времени»).

29.* В тексте главы мы создавали таблицу с помощью команды

```
CREATE TABLE databases ( is_open_source boolean, dbms_name text );
```

и заполняли ее данными.

```
INSERT INTO databases VALUES ( TRUE, 'PostgreSQL' );
INSERT INTO databases VALUES ( FALSE, 'Oracle' );
INSERT INTO databases VALUES ( TRUE, 'MySQL' );
INSERT INTO databases VALUES ( FALSE, 'MS SQL Server' );
```

Как вы думаете, являются ли все приведенные ниже команды равнозначными в смысле результатов, получаемых с их помощью?

```
SELECT * FROM databases WHERE NOT is_open_source;
SELECT * FROM databases WHERE is_open_source <> 'yes';
SELECT * FROM databases WHERE is_open_source <> 't';
SELECT * FROM databases WHERE is_open_source <> '1';
SELECT * FROM databases WHERE is_open_source <> 1;
```

30.* Обратимся к таблице, создаваемой с помощью команды

```
CREATE TABLE test_bool ( a boolean, b text );
```

Как вы думаете, какие из приведенных ниже команд содержат ошибку?

```
INSERT INTO test_bool VALUES ( TRUE, 'yes' );
INSERT INTO test_bool VALUES ( yes, 'yes' );
INSERT INTO test_bool VALUES ( 'yes', true );
INSERT INTO test_bool VALUES ( 'yes', TRUE );
INSERT INTO test_bool VALUES ( '1', 'true' );
INSERT INTO test_bool VALUES ( 1, 'true' );
INSERT INTO test_bool VALUES ( 't', 'true' );
INSERT INTO test_bool VALUES ( 't', truth );
INSERT INTO test_bool VALUES ( true, true );
INSERT INTO test_bool VALUES ( 1::boolean, 'true' );
INSERT INTO test_bool VALUES ( 111::boolean, 'true' );
```

Проверьте свои предположения практически, выполнив эти команды.

31.* Пусть в таблице birthdays хранятся даты рождения какой-то группы людей. Создайте эту таблицу с помощью команды

```
CREATE TABLE birthdays
( person text NOT NULL,
birthday date NOT NULL );
```

Добавьте в нее несколько строк, например:

```
INSERT INTO birthdays VALUES ( 'Ken Thompson', '1955-03-23' );
INSERT INTO birthdays VALUES ( 'Ben Johnson', '1971-03-19' );
INSERT INTO birthdays VALUES ( 'Andy Gibson', '1987-08-12' );
```

Давайте выберем из таблицы birthdays строки для всех людей, родившихся в каком-то конкретном месяце, например, в марте:

```
SELECT * FROM birthdays
  WHERE extract( 'mon' from birthday ) = 3;
```

В этой команде в вызове функции extract имеет место неявное приведение типов, т. к. ее вторым параметром должно быть значение типа timestamp. Полагаться на неявное приведение типов можно не всегда.

```
   person     |  birthday
--------------+------------
 Ken Thompson | 1955-03-23
 Ben Johnson  | 1971-03-19
(2 строки)
```

Если нам потребуется выяснить, кто из этих людей достиг возраста, скажем, 40 лет на момент выполнения запроса, то команда может быть такой (в последнем столбце показана дата достижения возраста 40 лет):

```
SELECT *, birthday + '40 years'::interval
  FROM birthdays
  WHERE birthday + '40 years'::interval < current_timestamp;
```

```
    person    |  birthday  |      ?column?
--------------+------------+---------------------
 Ken Thompson | 1955-03-23 | 1995-03-23 00:00:00
 Ben Johnson  | 1971-03-19 | 2011-03-19 00:00:00
(2 строки)
```

Можно заменить current_timestamp на current_date:

```
SELECT *, birthday + '40 years'::interval
  FROM birthdays
  WHERE birthday + '40 years'::interval < current_date;
```

А вот если мы захотим определить точный возраст каждого человека на текущий момент времени, то как получить этот результат? Первый вариант таков:

```
SELECT *, ( current_date::timestamp -
birthday::timestamp )::interval
FROM birthdays;
```

```
    person    |  birthday  |  interval
--------------+------------+------------
 Ken Thompson | 1955-03-23 | 22477 days
 Ben Johnson  | 1971-03-19 | 16637 days
 Andy Gibson  | 1987-08-12 | 10647 days
(3 строки)
```

Этот вариант не дает результата, представленного в удобной форме: он показывает возраст в днях, а для пересчета числа дней в число лет нужны дополнительные действия. Хотя, наверное, возможны ситуации, когда требуется определить возраст именно в днях.

В PostgreSQL предусмотрена специальная функция, позволяющая решить нашу задачу простым способом. Самостоятельно найдите ее описание в документации (см. раздел 9.9 «Операторы и функции даты/времени») и напишите команду с ее использованием.

32. Изучая приемы работы с массивами, можно, как и в других случаях, пользоваться способностью команды SELECT обходиться без создания таблиц. Покажем лишь два примера. Для объединения (конкатенации) массивов служит функция array_cat:

```
SELECT array_cat( ARRAY[ 1, 2, 3 ], ARRAY[ 3, 5 ] );
```

```
 array_cat
-------------
 {1,2,3,3,5}
(1 строка)
```

Удалить из массива элементы, имеющие указанное значение, можно таким образом:

```
SELECT array_remove( ARRAY[ 1, 2, 3 ], 3 );
```

```
 array_remove
--------------
 {1,2}
(1 строка)
```

Для работы с массивами предусмотрено много различных функций и операторов, представленных в разделе документации 9.18 «Функции и операторы для работы с массивами». Самостоятельно ознакомьтесь с ними, используя описанную технологию работы с командой SELECT.

33.* В разделе документации 8.15 «Массивы» сказано, что массивы могут быть многомерными и в них могут содержаться значения любых типов. Давайте сначала рассмотрим одномерные массивы *текстовых* значений.

Предположим, что пилоты авиакомпании имеют возможность высказывать свои пожелания насчет конкретных блюд, из которых должен состоять их обед во время полета. Для учета пожеланий пилотов необходимо модифицировать таблицу pilots, с которой мы работали в разделе 4.5.

```
CREATE TABLE pilots
( pilot_name text,
  schedule integer[],
  meal text[]
);
```

Добавим строки в таблицу:

```
INSERT INTO pilots
VALUES ( 'Ivan', '{ 1, 3, 5, 6, 7 }'::integer[],
         '{ "сосиска", "макароны", "кофе" }'::text[] ),
       ( 'Petr', '{ 1, 2, 5, 7 }'::integer [],
         '{ "котлета", "каша", "кофе" }'::text[] ),
       ( 'Pavel', '{ 2, 5 }'::integer[],
         '{ "сосиска", "каша", "кофе" }'::text[] ),
       ( 'Boris', '{ 3, 5, 6 }'::integer[],
         '{ "котлета", "каша", "чай" }'::text[] );
```

```
INSERT 0 4
```

Обратите внимание, что каждое из текстовых значений, включаемых в литерал массива, заключается в двойные кавычки, а в качестве типа данных указывается text[].

Вот что получилось:

```
SELECT * FROM pilots;
```

```
 pilot_name |  schedule   |        meal
------------+-------------+-------------------------
 Ivan       | {1,3,5,6,7} | {сосиска,макароны,кофе}
 Petr       | {1,2,5,7}   | {котлета,каша,кофе}
 Pavel      | {2,5}       | {сосиска,каша,кофе}
 Boris      | {3,5,6}     | {котлета,каша,чай}
(4 строки)
```

Давайте получим список пилотов, предпочитающих на обед сосиски:

```
SELECT * FROM pilots WHERE meal[ 1 ] = 'сосиска';
```

```
 pilot_name |   schedule  |         meal
------------+-------------+-------------------------
 Ivan       | {1,3,5,6,7} | {сосиска,макароны,кофе}
 Pavel      | {2,5}       | {сосиска,каша,кофе}
(2 строки)
```

Предположим, что руководство авиакомпании решило, что пища пилотов должна быть разнообразной. Оно позволило им выбрать свой рацион на каждый из четырех дней недели, в которые пилоты совершают полеты. Для нас это решение руководства выливается в необходимость модифицировать таблицу, а именно: столбец meal теперь будет содержать двумерные массивы. Определение этого столбца станет таким:

```
meal text[][]
```

**Задание.** Создайте новую версию таблицы и соответственно измените команду INSERT, чтобы в ней содержались литералы *двумерных* массивов. Они будут выглядеть примерно так:

```
'{ { "сосиска", "макароны", "кофе" },
   { "котлета", "каша", "кофе" },
   { "сосиска", "каша", "кофе" },
   { "котлета", "каша", "чай" } }'::text[][]
```

Сделайте ряд выборок и обновлений строк в этой таблице. Для обращения к элементам двухмерного массива нужно использовать два индекса. Не забывайте, что по умолчанию номера индексов начинаются с единицы.

34. В тексте раздела 4.6 мы выполняли обновление JSON-объекта с помощью функции jsonb_set: добавляли значение в массив. Для обновления скалярных значений, например, по ключу trips, можно сделать так:

```
UPDATE pilot_hobbies
    SET hobbies = jsonb_set( hobbies, '{ trips }', '10' )
    WHERE pilot_name = 'Pavel';
```

```
UPDATE 1
```

Второй параметр функции — это путь в пределах JSON-объекта. Он теперь представляет собой лишь имя ключа. Однако его необходимо заключить в фигурные скобки. Третий параметр — это новое значение. Хотя оно числовое, но все равно требуется записать его в одинарных кавычках.

```
SELECT pilot_name, hobbies->'trips' AS trips FROM pilot_hobbies;
```

```
 pilot_name | trips
------------+-------
 Ivan       | 3
 Petr       | 2
 Boris      | 0
 Pavel      | 10
(4 строки)
```

**Задание.** Самостоятельно выполните изменение значения по ключу home_lib в одной из строк таблицы.

35. Изучая приемы работы с типами JSON, можно, как и в случае с массивами, пользоваться способностью команды SELECT обходиться без создания таблиц. Покажем лишь один пример. Для добавления нового ключа и соответствующего ему значения в уже существующий объект, можно воспользоваться оператором «||»:

```
SELECT '{ "sports": "хоккей" }'::jsonb || '{ "trips": 5 }'::jsonb;
```

```
             ?column?
---------------------------------
 {"trips": 5, "sports": "хоккей"}
(1 строка)
```

Для работы с типами JSON предусмотрено много различных функций и операторов, представленных в разделе документации 9.15 «Функции и операторы JSON». Самостоятельно ознакомьтесь с ними, используя описанную технологию работы с командой SELECT.

36.* Объекты JSON в разных строках таблицы могут иметь различные наборы ключей. Добавьте дополнительный ключ и соответствующее ему значение в JSON-объект какой-нибудь строки таблицы pilots. Воспользуйтесь оператором «||».

37. Объекты JSON позволяют не только добавлять в них новые ключи, но также и удалять из них ключи существующие. Удалите один из ключей из JSON-объекта какой-нибудь строки таблицы pilots. Соответствующее ему значение будет также удалено, т. к. без ключа оно не может существовать. Воспользуйтесь оператором «—».

# 5 Основы языка определения данных

Как мы уже говорили ранее, язык SQL традиционно разделяется на две группы команд. Первая из них предназначена для определения данных, т. е. для создания объектов базы данных, таких, например, как таблицы. Вторая группа команд служит для выполнения различных операций с данными, таких, как вставка строк в таблицы, выполнение запросов к ним, обновление и удаление строк из таблиц. В этой главе мы сосредоточимся на командах первой группы, т. е. на определении данных. Рассмотрим все таблицы базы данных «Авиаперевозки».

## 5.1 Значения по умолчанию и ограничения целостности

В последующих параграфах этой главы в качестве «опорной» базы данных мы будем использовать базу данных «Авиаперевозки», описанную в первой главе. Однако основные сведения о значениях по умолчанию и ограничениях мы проиллюстрируем на той простой базе данных, состоящей из двух таблиц — «Студенты» и «Успеваемость», о которой речь шла также в первой главе пособия.

Сначала представим описание таблицы «Студенты» (students). Она имеет следующую структуру (т. е. набор атрибутов и их типы данных):

| Описание атрибута | Имя атрибута | Тип данных | Тип PostgreSQL | Ограничения |
|---|---|---|---|---|
| № зачетной книжки | record_book | Числовой | numeric(5) | NOT NULL |
| Ф. И. О. | name | Символьный | text | NOT NULL |
| Серия документа | doc_ser | Числовой | numeric(4) | |
| Номер документа | doc_num | Числовой | numeric(6) | |

Для атрибута «Серия документа, удостоверяющего личность» мы выбрали числовой тип, хотя, пожалуй, более дальновидным был бы выбор символьного типа (см. задание 10 в конце главы).

Теперь перейдем к таблице «Успеваемость» (progress). Структура ее такова:

| Описание атрибута | Имя атрибута | Тип данных | Тип PostgreSQL | Ограничения |
|---|---|---|---|---|
| № зачетной книжки | record_book | Числовой | numeric(5) | NOT NULL |
| Учебная дисциплина | subject | Символьный | text | NOT NULL |
| Учебный год | acad_year | Символьный | text | NOT NULL |
| Семестр | term | Числовой | numeric(1) | NOT NULL<br>term = 1 OR term = 2 |
| Оценка | mark | Числовой | numeric(1) | DEFAULT 5<br>mark >= 3 AND<br>mark <= 5 |

В описаниях таблиц «Студенты» и «Успеваемость» есть колонки «Ограничения». Они уже содержат конкретные сведения, хотя ограничения мы еще детально не рассматривали. Таким образом, мы немного забежали вперед, но по мере рассмотрения ограничений вам будет становиться понятно назначение каждого из них в обеих таблицах.

Итак, начнем со **значений по умолчанию**. При работе с базами данных нередко возникают ситуации, когда то или иное значение является типичным для какого-то конкретного столбца. Например, если мы при проектировании таблицы «Успеваемость» (progress), предназначенной для учета успеваемости студентов, знаем, что их успехи, как правило, заслуживают оценки «отлично», то в команде CREATE TABLE мы можем отразить этот факт с помощью ключевого слова DEFAULT:

```
CREATE TABLE progress
...
    mark numeric( 1 ) DEFAULT 5,
...
```

Теперь перейдем к рассмотрению различных видов ограничений (constraints). Будем придерживаться того же порядка, в котором они представлены в документации.

Первым будет **ограничение CHECK**. Для его создания используется ключевое слово CHECK, за которым следует выражение в круглых скобках, содержащее одно или несколько условий, налагаемых на атрибуты таблицы. Это ограничение бывает двух видов: ограничение уровня атрибута и уровня таблицы. Различие между ними только в синтаксическом оформлении: в обоих случаях в выражении могут содержаться обращения не только к одному, но также и к нескольким атрибутам таблицы. В первом случае ограничение CHECK является частью определения одного конкретного атрибута, а во втором случае оно записывается как самостоятельный элемент определения таблицы. Каждое ограничение имеет имя. Мы можем задать его сами с помощью ключевого слова CONSTRAINT. Если же мы этого не сделаем, тогда СУБД сформирует имя автоматически. Когда мы задаем имя сами, мы можем выбрать его с учетом сути налагаемых ограничений, с позиции предметной области. Если же это имя формирует СУБД, оно будет сформировано «механически», т. к. СУБД не знает ни сути этих ограничений, ни специфики предметной области.

В качестве примера приведем ограничения, налагаемые на атрибуты term и mark из таблицы «Успеваемость» (progress). Семестр может иметь только два значения: 1 и 2. Отметка фактически может иметь только три значения: 3, 4 или 5.

```
CREATE TABLE progress
( ...
    term numeric( 1 ) CHECK ( term = 1 OR term = 2 ),
    mark numeric( 1 ) CHECK ( mark >= 3 AND mark <= 5 ),
...
);
```

В данном случае можно и не давать этим ограничениям какие-либо специфические имена, поскольку суть этих ограничений очевидна. Тем не менее, поскольку имена ограничений используются в тех сообщениях, которые выводит СУБД при попытке нарушения ограничений, все же можно придумать для них осмысленные имена, которые облегчат понимание причин появления сообщений об ошибках.

В качестве примера приведем ограничение на допустимые значения атрибута mark, а оформим его как ограничение уровня таблицы:

```
CREATE TABLE progress
( ...
   mark numeric( 1 ),
   CONSTRAINT valid_mark CHECK ( mark >= 3 AND mark <= 5 ),
```

```
...
);
```

Следующим видом ограничений, который мы рассмотрим, будет **NOT NULL**. Оно означает, что в столбце таблицы, на который наложено это ограничение, должны обязательно присутствовать какие-либо определенные значения. При разработке баз данных, исходя из логики конкретной предметной области, зачастую требуется использовать это ограничение. Как сказано в документации, оно функционально эквивалентно ограничению CHECK ( column_name IS NOT NULL ), но в PostgreSQL создание явного ограничения NOT NULL является более эффективным подходом.

Еще один вид ограничений — это **ограничение уникальности** UNIQUE. Такое ограничение, наложенное на конкретный столбец, означает, что все значения, содержащиеся в этом столбце в различных строках таблицы, должны быть уникальными, т. е. не должны повторяться. Ограничение уникальности может включать в себя и несколько столбцов. В этом случае уникальной должна быть уже комбинация их значений.

Когда в ограничение уникальности включается только один столбец, то можно задать ограничение непосредственно в определении столбца. Например, для таблицы «Студенты» (students) было бы логично потребовать, чтобы уникальными были значения столбца record_book:

```
CREATE TABLE students
( record_book numeric( 5 ) UNIQUE,
...
```

Это ограничение можно было бы записать и так, дав ему осмысленное имя:

```
CREATE TABLE students
( record_book numeric( 5 ),
  name text NOT NULL,
...
  CONSTRAINT unique_record_book UNIQUE ( record_book ),
...
);
```

Опять обратимся к таблице «Студенты» (students) и покажем, как можно создать ограничение уникальности, включающее более одного столбца. В этой таблице первичным ключом является столбец record_book, но очевидно, что комбинация значений серии и номера документа, удостоверяющего личность, является уникальной. Поэтому можно модифицировать определение таблицы таким образом:

```
CREATE TABLE students
( ...
    doc_ser numeric( 4 ),
    doc_num numeric( 6 ),
...
    CONSTRAINT unique_passport UNIQUE ( doc_ser, doc_num ),
...
);
```

При добавлении ограничения уникальности автоматически создается индекс на основе B-дерева для поддержки этого ограничения.

Переходим к **первичным ключам**. Как мы уже говорили ранее, этот ключ является уникальным идентификатором строк в таблице. Ключ может быть как простым, т. е. включать только один атрибут, так и составным, т. е. включать более одного атрибута. При этом в отличие от уникального ключа, определяемого с помощью ограничения UNIQUE, атрибуты, входящие в состав первичного ключа, не могут иметь значений NULL. Таким образом, определение первичного ключа эквивалентно определению уникального ключа, дополненного ограничением NOT NULL. Однако не стоит в реальной работе заменять первичный ключ комбинацией ограничений UNIQUE и NOT NULL, поскольку теория баз данных требует наличия в каждой таблице именно первичного ключа. Первичный ключ является частью метаданных, его наличие позволяет другим таблицам использовать его в качестве уникального идентификатора строк в данной таблице. Это удобно, например, при создании внешних ключей, речь о которых пойдет ниже. Перечисленными свойствами обладает также и уникальный ключ.

Если первичный ключ состоит из одного атрибута, то можно указать его непосредственно в определении этого атрибута:

```
CREATE TABLE students
( record_book numeric( 5 ) PRIMARY KEY,
...
);
```

А можно сделать это и в виде отдельного ограничения:

```
CREATE TABLE students
( record_book numeric( 5 ),
...
  PRIMARY KEY ( record_book )
);
```

В случае создания составного первичного ключа имена столбцов, входящих в его состав, перечисляются в выражении PRIMARY KEY через запятую:

```
PRIMARY KEY ( column1, column2, ...)
```

При добавлении первичного ключа автоматически создается индекс на основе B-дерева для поддержки этого ограничения.

В таблице может быть любое число ограничений UNIQUE, дополненных ограничением NOT NULL, но первичный ключ может быть только один. PostgreSQL допускает и отсутствие первичного ключа, хотя строгая теория реляционных баз данных не рекомендует так поступать.

Завершаем наш обзор различных видов ограничений рассмотрением такого важного понятия, как **внешний ключ** (foreign key). Внешние ключи являются средством поддержания так называемой **ссылочной целостности** (referential integrity) между связанными таблицами. Напомним, что это означает, на примере таблиц «Студенты» (students) и «Успеваемость» (progress). В первой из них содержатся данные о студентах, а во второй — сведения об их успеваемости. Поскольку в процессе обучения студенты сдают целый ряд зачетов и экзаменов, то в таблице «Успеваемость» для каждого студента может присутствовать несколько строк. Для большинства из них это так

и будет, хотя, в принципе, возможна ситуация, когда для какого-то студента в таблице «Успеваемость» не окажется ни одной строки (если, он, например, находится в академическом отпуске).

Конечно, должна быть возможность определить, какому студенту принадлежат те или иные оценки, т. е. какие строки в таблице «Успеваемость» с какими строками в таблице «Студенты» связаны. Для решения этой задачи не требуется в каждой строке таблицы «Успеваемость» повторять все сведения о студенте: номер зачетной книжки, фамилию, имя и отчество, данные документа, удостоверяющего личность. Достаточно включить в состав каждой строки таблицы «Успеваемость» лишь уникальный идентификатор строки из таблицы «Студенты». В нашем случае это будет номер зачетной книжки — record_book. Данный атрибут и будет являться внешним ключом таблицы «Успеваемость». Таким образом, получив строку из таблицы «Студенты», можно будет найти все соответствующие ей строки в таблице «Успеваемость», сопоставив значения атрибутов record_book в строках обеих таблиц. В результате мы сможем получить все строки таблицы «Успеваемость», связанные с конкретной строкой из таблицы «Студенты» по внешнему ключу.

Таблица «Успеваемость» будет **ссылающейся** (referencing), а таблица «Студенты» — **ссылочной** (referenced). Обратите внимание, что внешний ключ ссылающейся таблицы ссылается на первичный ключ ссылочной таблицы. Допускается ссылка и на уникальный ключ, не являющийся первичным. В данном контексте для описания отношений между таблицами можно сказать, что таблица students является **главной**, а таблица progress — **подчиненной**.

Создать внешний ключ можно в формате ограничения уровня атрибута следующим образом:

```
CREATE TABLE progress
( record_book numeric( 5 ) REFERENCES students ( record_book ),
...
);
```

Предложение REFERENCES создает ограничение ссылочной целостности и указывает в качестве ссылочного ключа атрибут record_book. Это означает, что в таблицу «Успеваемость» (progress) нельзя ввести строку, значение атрибута record_book которой отсутствует в таблице «Студенты» (students). Говоря простым языком, нельзя ввести запись об оценке того студента, информация о котором еще не введена в таблицу «Студенты».

Поскольку внешний ключ в нашем примере ссылается на первичный ключ, можно использовать сокращенную форму записи этого ограничения, не указывая список атрибутов:

```
CREATE TABLE progress
( record_book numeric( 5 ) REFERENCES students,
...
);
```

Можно определить внешний ключ и в форме ограничения уровня таблицы:

```
CREATE TABLE progress
( record_book numeric( 5 ),
...
```

```
  FOREIGN KEY ( record_book )
    REFERENCES students ( record_book )
);
```

Конечно, число атрибутов и их типы данных во внешнем ключе ссылающейся таблицы и в первичном ключе ссылочной таблицы должны быть согласованы.

Ограничению внешнего ключа можно присвоить наименование, как и любому другому ограничению, с помощью ключевого слова CONSTRAINT.

При наличии связей между таблицами, организованных с помощью внешних ключей, необходимо придерживаться определенной политики при выполнении операций удаления и обновления строк в ссылочных таблицах, т. е. в тех, на которые ссылаются другие таблицы. В нашем примере ситуация принятия «политического» решения возникает при удалении строк из таблицы «Студенты» (students). Конечно, если бы было принято решение хранить всю историю успеваемости студентов, в том числе и отчисленных, тогда строки из таблицы students вообще не удалялись бы. Но, упрощая реальную ситуацию, мы решили историю не хранить. Тогда возникает закономерный вопрос: что делать со строками в таблице «Успеваемость» (progress), которые ссылаются на удаляемую строку в таблице «Студенты» (students)? Возможны несколько вариантов.

1. Удаление связанных строк из таблицы «Успеваемость» (progress), что означает, что при отчислении студента будет удаляться вся история его успехов в учебе. Эта операция называется каскадным удалением и для ее реализации в определение внешнего ключа добавляются ключевые слова ON DELETE CASCADE. Например:

   ```
   CREATE TABLE progress
   ( record_book numeric( 5 ),
   ...
     FOREIGN KEY ( record_book )
       REFERENCES students ( record_book )
       ON DELETE CASCADE
   );
   ```

2. Запрет удаления строки из таблицы «Студенты» (students), если в таблице «Успеваемость» (progress) есть хотя бы одна строка, ссылающаяся на удаляемую строку в таблице «Студенты». Для реализации такой политики в определение внешнего ключа добавляются ключевые слова ON DELETE RESTRICT или ON DELETE NO ACTION. Если в определении внешнего ключа не предписано конкретное действие, то по умолчанию используется NO ACTION. Оба эти варианта означают, что если в ссылающейся таблице, т. е. «Успеваемость», есть строки, ссылающиеся на удаляемую строку в таблице «Студенты», то операция удаления будет отменена, и будет выведено сообщение об ошибке. Отличие между этими двумя вариантами лишь в том, что при использовании NO ACTION можно отложить проверку выполнения ограничения на более поздний строк в рамках транзакции, а в случае RESTRICT проверка выполняется немедленно. Поэтому если бы внешний ключ определили таким образом:

   ```
   CREATE TABLE progress
   ( record_book numeric( 5 ),
   ...
   ```

```
  FOREIGN KEY ( record_book )
    REFERENCES students ( record_book )
    ON DELETE RESTRICT
);
```

или таким:

```
CREATE TABLE progress
( record_book numeric( 5 ),
...
  FOREIGN KEY ( record_book )
    REFERENCES students ( record_book )
);
```

то при попытке удаления строки из таблицы «Студенты» и наличии в таблице «Успеваемость» строк, связанных с ней, операция удаления была бы отменена с выводом сообщения об ошибке.

3. Присваивание атрибутам внешнего ключа в строках таблицы «Успеваемость» значения NULL. Для реализации этого подхода необходимо, чтобы на атрибуты внешнего ключа не было наложено ограничение NOT NULL. Оформляется этот вариант так:

```
CREATE TABLE progress
( record_book numeric( 5 ),
...
  FOREIGN KEY ( record_book )
    REFERENCES students ( record_book )
    ON DELETE SET NULL
);
```

4. Присваивание атрибутам внешнего ключа в строках таблицы «Успеваемость» (progress) значения DEFAULT, если оно, конечно, было предписано при создании таблицы. Оформляется этот вариант так (значение во фразе DEFAULT взято произвольное):

```
CREATE TABLE progress
( record_book numeric( 5 ) DEFAULT 12345,
...
  FOREIGN KEY ( record_book )
    REFERENCES students ( record_book )
    ON DELETE SET DEFAULT
);
```

Важно учитывать, что если в ссылочной таблице нет строки с тем же значением ключевого атрибута, которое было предписано во фразе DEFAULT при создании ссылающейся таблицы, то будет иметь место нарушение ограничения ссылочной целостности и операция удаления не будет выполнена.

При выполнении операции UPDATE используются эти же варианты подходов по отношению к обеспечению ссылочной целостности. Аналогом каскадного удаления является каскадное обновление:

```
CREATE TABLE progress
( record_book numeric( 5 ),
...
  FOREIGN KEY ( record_book )
    REFERENCES students ( record_book )
    ON UPDATE CASCADE
);
```

В этом случае измененные значения ссылочных атрибутов копируются в ссылающиеся строки ссылающейся таблицы, т. е. новое значение атрибута record_book из строки таблицы «Студенты» будет скопировано во все строки таблицы «Успеваемость», ссылающиеся на обновленную строку.

После рассмотрения всех видов ограничений целостности базы данных мы можем привести окончательные определения таблиц «Студенты» и «Успеваемость». Окончательными они являются лишь в том смысле, что именно их нужно брать за основу при выполнении заданий, приведенных в конце главы. Эти определения ни в коем случае не являются идеальными, эталонными. Выполняя задания, вы это увидите сами.

Прежде чем создавать таблицы, создайте базу данных edu:

```
createdb -U postgres edu
```

Подключитесь к ней:

```
psql -d edu -U postgres
```

Создайте обе таблицы:

```
CREATE TABLE students
( record_book numeric( 5 ) NOT NULL,
  name text NOT NULL,
  doc_ser numeric( 4 ),
  doc_num numeric( 6 ),
  PRIMARY KEY ( record_book )
);

CREATE TABLE progress
( record_book numeric( 5 ) NOT NULL,
  subject text NOT NULL,
  acad_year text NOT NULL,
  term numeric( 1 ) NOT NULL CHECK ( term = 1 OR term = 2 ),
  mark numeric( 1 ) NOT NULL CHECK ( mark >= 3 AND mark <= 5 )
    DEFAULT 5,
  FOREIGN KEY ( record_book )
    REFERENCES students ( record_book )
    ON DELETE CASCADE
    ON UPDATE CASCADE
);
```

## 5.2 Создание и удаление таблиц

Настало время познакомить вас с оставшимися таблицами базы данных «Авиаперевозки». Рекомендуем вам, прежде чем приступать к дальнейшей работе, освежить в памяти описание этой предметной области, приведенное в главе 1.

Поскольку в главе 3 вы уже создавали таблицы с помощью команды CREATE TABLE, то мы изберем такую стратегию: мы представим вам команды для создания оставшихся таблиц, но выполнять эти команды уже не требуется, достаточно будет только просматривать описания таблиц с помощью команды \d утилиты psql.

В главе 3 мы сначала представляли описание каждой создаваемой таблицы: назначения и имена столбцов, их типы данных и ограничения, которым каждый столбец должен удовлетворять. А уже после этого мы переходили непосредственно к SQL-команде создания таблицы в базе данных — CREATE TABLE. Но в этой главе мы, во избежание повторов, сразу будем показывать команды CREATE TABLE, дополняя их подробными комментариями.

Если вы еще не запустили утилиту psql, то запустите ее и подключитесь к базе данных demo с учетной записью пользователя СУБД с именем postgres:

```
psql -d demo -U postgres
```

Выберите в качестве текущей схемы схему bookings:

```
SET search_path TO bookings;
```

Если вы уже были подключены к другой базе данных, то можете, не выходя из утилиты psql, просто подключиться к нужной вам базе данных с помощью команды \connect. Напомним, что команды, имена которых начинаются с символа «\», не являются SQL-командами, а являются командами утилиты psql. Конечно, за этими короткими командами, например, \d, могут скрываться сложные SQL-запросы к системным таблицам базы данных. Просто утилита psql избавляет пользователя от необходимости вводить эти сложные запросы. Для подключения к базе данных demo изнутри psql сделайте так:

```
\connect demo
```

Существует и сокращенный вариант этой команды:

```
\c demo
```

При создании таблиц необходимо учитывать связи между ними. Поэтому сначала должны создаваться ссылочные таблицы, а потом — ссылающиеся. Конечно, возможна ситуация, когда образуется иерархия таблиц. Таблица, находящаяся в середине такой иерархии, выполняет обе роли: ссылающейся и ссылочной таблицы. Тогда нужно продвигаться «вниз» от вершины иерархии, где находится таблица, не имеющая внешних ключей.

Если в базе данных нет циклических ссылок таблиц друг на друга, то всегда существует таблица (или таблицы), которые не ссылаются ни на какие другие таблицы. С них и нужно начинать создание базы данных. Перед созданием очередной таблицы, имеющей внешний ключ (ключи), уже должны существовать все ссылочные таблицы для нее.

При наличии циклических ссылок таблиц друг на друга придется воспользоваться командой ALTER TABLE, о которой речь пойдет в следующем разделе этой главы.

Поскольку две первые таблицы — «Самолеты» (aircrafts) и «Места» (seats) — мы уже изучили в главе 3, то перейдем к таблице **«Аэропорты» (airports)**. Она не имеет внешних ключей.

В этой таблице в качестве кода аэропорта служат трехбуквенные обозначения, утвержденные специальной организацией. При этом используются только буквы латинского алфавита. Каждый аэропорт имеет также и полное название. Оно не всегда совпадает с названием города, в котором аэропорт находится. Для города не предусмотрено отдельной сущности, поэтому таблицы городов нет. Однако название города присутствует в виде атрибута city. Назначение остальных атрибутов ясно из комментариев, приведенных в SQL-команде.

Комментарии в языке SQL обозначаются двумя символами «дефис». При создании таблиц в среде утилиты psql вводить комментарии не нужно, но если вы создаете текстовый файл, содержащий команды для создания объектов базы данных, то комментарии нужно ввести. Это сделает такой файл более понятным для вас в будущем.

```
CREATE TABLE airports
( airport_code char( 3 ) NOT NULL, -- Код аэропорта
  airport_name text NOT NULL,      -- Название аэропорта
  city text NOT NULL,              -- Город
  longitude float NOT NULL, -- Координаты аэропорта: долгота
  latitude float NOT NULL,  -- Координаты аэропорта: широта
  timezone text NOT NULL,   -- Часовой пояс аэропорта
  PRIMARY KEY ( airport_code )
);
```

Посмотрите описание этой таблицы:

```
\d airports
```

В команде \d можно было ввести лишь первые символы имени таблицы и нажать клавишу Tab — psql дополнил бы имя. При этом символов должно быть столько, чтобы они однозначно определяли имя таблицы. В нашем случае есть еще таблица aircrafts, поэтому можно было сделать так:

```
\d airp
```

а затем нажать клавишу Tab. Можно было использовать автодополнение с самого начала: введя только первую букву имени таблицы, т. е. «a», сразу нажать Tab — psql дополнит до «air», поскольку есть варианты aircrafts и airports. Дальше вы можете добавить букву «p» и нажать Tab, а можете сначала просмотреть возможные варианты, нажав Tab дважды подряд.

В результате вы получите примерно такой вывод на экран:

```
Таблица "bookings.airports"
   Столбец    |       Тип        | Модификаторы
--------------+------------------+--------------
 airport_code | character(3)     | NOT NULL
 airport_name | text             | NOT NULL
 city         | text             | NOT NULL
```

```
 longitude    | double precision | NOT NULL
 latitude     | double precision | NOT NULL
 timezone     | text             | NOT NULL
Индексы:
    "airports_pkey" PRIMARY KEY, btree (airport_code)
Ссылки извне:
    TABLE "flights" CONSTRAINT "flights_arrival_airport_fkey"
      FOREIGN KEY (arrival_airport)
      REFERENCES airports(airport_code)
    TABLE "flights" CONSTRAINT "flights_departure_airport_fkey"
      FOREIGN KEY (departure_airport)
      REFERENCES airports(airport_code)
```

В этом выводе в выражении «bookings.airports» слово bookings означает имя **схемы**. Как мы уже говорили ранее, это, упрощенно говоря, раздел базы данных, в котором и создаются таблицы и другие объекты. По умолчанию используется схема public, но в базе данных demo создана схема bookings.

Поскольку мы задавали первичный ключ, то для его реализации был автоматически создан индекс. Имя индекса в наше случае — airports_pkey. Оно было сгенерировано ядром PostgreSQL. Указан также и тип индекса — btree, т. е. B-дерево. Далее в круглых скобках приводится список ключевых атрибутов. В нашем случае он состоит из одного атрибута — airport_code.

Обратите внимание, что в команде создания таблицы «Аэропорты» (airports) мы указывали для атрибутов longitude и latitude тип данных float, определенный в стандарте SQL. Однако, согласно документации, если при объявлении типа float параметр, задающий точность, не указан, то это будет равносильно использованию типа double precision.

PostgreSQL предлагает свое расширение — команду COMMENT, которая позволяет создавать комментарии (описания) к различным объектам базы данных. Эти комментарии будут также сохраняться в базе данных. Например, для создания описания столбца city таблицы airports нужно сделать так:

```
COMMENT ON COLUMN airports.city IS 'Город';
```

Чтобы увидеть описания столбцов таблицы, нужно в команде \d добавить символ «+», например:

```
\d+ airports
```

Следующая таблица — **«Рейсы» (flights)**. Назначение ее атрибутов должно быть в целом понятно из комментариев, присутствующих в SQL-команде. Сначала приведем саму команду, а затем сделаем ряд пояснений.

```
CREATE TABLE flights
( flight_id           serial NOT NULL,    -- Идентификатор рейса
  flight_no           char( 6 ) NOT NULL, -- Номер рейса

  scheduled_departure timestamptz NOT NULL, -- Время вылета
                                            -- по расписанию
  scheduled_arrival   timestamptz NOT NULL, -- Время прилета
                                            -- по расписанию
  departure_airport   char( 3 ) NOT NULL, -- Аэропорт отправления
```

```
  arrival_airport     char( 3 ) NOT NULL, -- Аэропорт прибытия
  status              varchar( 20 ) NOT NULL, -- Статус рейса
  aircraft_code       char( 3 ) NOT NULL, -- Код самолета, IATA
  actual_departure   timestamptz,      -- Фактическое время вылета
  actual_arrival     timestamptz,      -- Фактическое время прилета
  CHECK ( scheduled_arrival > scheduled_departure ),
  CHECK ( actual_arrival IS NULL OR
     ( actual_departure IS NOT NULL AND
       actual_arrival IS NOT NULL AND
       actual_arrival > actual_departure
     )
  ),
  CHECK ( status IN ( 'On Time', 'Delayed', 'Departed',
                      'Arrived', 'Scheduled', 'Cancelled' )
  ),
  PRIMARY KEY ( flight_id ),
  UNIQUE ( flight_no, scheduled_departure ),
  FOREIGN KEY ( aircraft_code )
      REFERENCES aircrafts ( aircraft_code ),
  FOREIGN KEY ( arrival_airport )
      REFERENCES airports ( airport_code ),
  FOREIGN KEY ( departure_airport )
      REFERENCES airports ( airport_code )
);
```

В таблице предусмотрено три внешних ключа, которые ссылаются на таблицы «Самолеты» (aircrafts) и «Аэропорты» (airports). В качестве первичного ключа используется так называемый **суррогатный ключ**, состоящий из одного атрибута — flight_id. Обратите внимание, что тип данных этого атрибута — serial, т. е. значения целого типа для этого атрибута будут извлекаться из последовательности. Суррогатный ключ — это уникальный ключ, назначение которого — только идентифицировать строки в таблице. Зачастую для него используются целочисленные значения. Такому ключу не соответствует никакое свойство никакой сущности реального мира. Это — абстракция, позволяющая в ряде случаев упростить определения таблиц, например, за счет сокращения числа атрибутов во внешних ключах до одного. В нашей таблице «Рейсы» (flights) суррогатный ключ как раз и служит для того, чтобы в таблицах, ссылающихся на нее, внешние ключи состояли только из атрибута flight_id.

Конечно, существует и естественный уникальный ключ, состоящий из двух атрибутов: номер рейса (flight_no) и время вылета по расписанию (scheduled_departure). Для него нам придется создать уникальный ключ, чтобы избежать дублирования значений: очевидно, что в один и тот же момент времени не могут выполняться два (и более) рейса, имеющие один и тот же номер.

Обратите внимание, что для атрибутов, имеющих смысл даты/времени, выбран тип данных timestamptz, т. е. временная отметка с указанием часового пояса. Это важно, т. к. перелеты могут совершаться между городами, находящимися в разных часовых поясах, а время вылета и время прилета указывается местное.

Поясним смысл каждого из трех ограничений CHECK. Первое ограничение говорит о том, что время прилета по расписанию должно быть больше времени вылета по расписанию. Это представляется очевидным, т. к. длительность полета всегда больше нуля.

Второе ограничение более сложное. Его можно условно разделить на две части, соединенные логической операцией «ИЛИ». Первая часть говорит о том, что если самолет *еще не прилетел* (т. е. значение actual_arrival равно NULL), то фактическое время вылета нас, образно говоря, не интересует. Самолет мог еще не вылететь или уже вылететь. Но даже если он уже и вылетел, и значение атрибута actual_departure отлично от NULL, то все равно сравнить его со значением атрибута actual_arrival, которое пока еще равно NULL, невозможно. Речь идет о сравнении вида «>» или «<». Вторая часть этого ограничения должна гарантировать, что если самолет *уже прилетел*, то, во-первых, фактическое время вылета должно быть не равно NULL, а во-вторых, фактическое время прилета должно быть больше фактического времени вылета.

И наконец, третье ограничение CHECK ограничивает множество допустимых значений атрибута status следующим списком:

- Scheduled — рейс доступен для бронирования (это происходит за месяц до плановой даты вылета, а до этого запись о рейсе не существует в базе данных);
- On Time — рейс доступен для регистрации (за сутки до плановой даты вылета) и не задержан;
- Delayed — рейс доступен для регистрации (за сутки до плановой даты вылета), но задержан;
- Departed — самолет уже вылетел и находится в воздухе;
- Arrived — самолет прибыл в пункт назначения;
- Cancelled — рейс отменен.

Просмотреть описание таблицы в базе данных можно так:

```
\d flights
```

Поскольку до сих пор мы давали подробные пояснения по каждой таблице, то сейчас ограничимся только указанием на те сведения, которые могут быть непонятными. В частности, обратите внимание, что для атрибута flight_id указан тип данных integer, а не serial, как предписано в команде для создания этой таблицы. В главе 4 при рассмотрении типа данных serial мы говорили, ссылаясь на документацию, что этот тип является, по сути, удобной синтаксической заменой, избавляющей администратора базы данных от необходимости выполнения SQL-команд для явного создания последовательности и привязки ее к конкретному столбцу таблицы. О том, что значения для этого столбца будут формироваться с помощью последовательности, говорит фраза

```
DEFAULT nextval('flights_flight_id_seq'::regclass)
```

В этой фразе указано и имя последовательности — flights_flight_id_seq. Если выполнить команду

```
\d
```

то можно увидеть эту последовательность в списке объектов базы данных.

```
 Список отношений
  Схема   |         Имя           |        Тип         | Владелец
----------+-----------------------+--------------------+----------
...
 bookings | flights_flight_id_seq | последовательность | postgres
...
(11 строк)
```

Чтобы посмотреть описание последовательности flights_flight_id_seq, нужно использовать команду \d:

```
\d flights_flight_id_seq
```

В базе данных есть еще одна таблица, не имеющая внешних ключей, — **«Бронирования» (bookings)**. Это довольно простая таблица. В ней всего три атрибута. Атрибут «Номер бронирования» (book_ref) является первичным ключом. Поскольку он представляет собой шестизначную комбинацию латинских букв и цифр, то в качестве типа данных для него выбран тип character (сокращенно — char). Для атрибута «Дата бронирования» (book_date) выбран тип данных timestamptz — временная отметка с часовым поясом, т. к. билеты могут приобретаться в городах, находящихся в различных часовых поясах. В главе 4 мы уже говорили о том, что в случаях, требующих точных вычислений, необходимо использовать числа с фиксированной точностью. Работа с денежными суммами как раз и является одним из таких случаев. Поэтому для атрибута «Полная сумма бронирования» (total_amount) выбирается тип данных numeric, при этом масштаб, т. е. число цифр справа от десятичной точки (запятой), будет равен 2.

```
CREATE TABLE bookings
( book_ref     char( 6 )   NOT NULL,   -- Номер бронирования
  book_date    timestamptz NOT NULL,   -- Дата бронирования
  total_amount numeric( 10, 2 ) NOT NULL, -- Полная стоимость
                                          -- бронирования
  PRIMARY KEY ( book_ref )
);
```

С таблицей «Бронирования» (bookings) по внешнему ключу связана таблица **«Билеты» (tickets)**. В качестве первичного ключа служит атрибут «Номер билета» (ticket_no). Хотя уникальные тринадцатизначные номера билетов — числовые, но в них могут присутствовать лидирующие нули, поэтому числовой тип данных здесь не годится, а приходится использовать тип character (сокращенно — char). В качестве идентификатора пассажира будет использоваться номер документа, удостоверяющего личность, а номера таких документов могут содержать, например, лидирующие нули, поэтому атрибут «Идентификатор пассажира» (passenger_id) будет не числовым, а символьным — varchar. Атрибут «Имя пассажира» (passenger_name) содержит имя и фамилию пассажира, записанные заглавными латинскими буквами, а вот отчество не используется. Тип данных, конечно, text. Очень интересный атрибут «Контактные данные пассажира» (contact_data). Его особенность в том, что эти данные могут иметь некоторую структуру, но при этом создавать дополнительные атрибуты в таблице нецелесообразно. С такими данными — их называют полуструктурированными — PostgreSQL хорошо умеет работать: для них предусмотрены типы json и jsonb. В нашей таблице используется тип jsonb и вот почему: хотя ввод данных такого типа несколько замедляется из-за необходимости выполнения разбора данных,

но этот разбор выполняется однократно, только при вводе, а последующая обработка уже разобранных данных ускоряется. Подробно типы json и jsonb рассмотрены в главе 4.

Внешним ключом будет атрибут «Номер бронирования» (book_ref), поскольку в рамках каждой процедуры бронирования может быть оформлено более одного билета.

```
CREATE TABLE tickets
( ticket_no      char( 13 ) NOT NULL,    -- Номер билета
  book_ref       char( 6 ) NOT NULL,     -- Номер бронирования
  passenger_id   varchar( 20 ) NOT NULL, -- Идентификатор пассажира
  passenger_name text NOT NULL,          -- Имя пассажира
  contact_data   jsonb,              -- Контактные данные пассажира
  PRIMARY KEY ( ticket_no ),
  FOREIGN KEY ( book_ref )
      REFERENCES bookings ( book_ref )
);
```

Информация обо всех перелетах сохраняется в таблице **«Перелеты» (ticket_flights)**. Перелет — это перемещение конкретного пассажира из одного города в другой на конкретном авиарейсе. Перелеты вписываются в электронные билеты, при этом в каждый электронный билет может быть вписано более одного перелета. Поэтому первичным ключом будет комбинация двух атрибутов: «Номер билета» (ticket_no) и «Идентификатор рейса» (flight_id). С каждым перелетом связан класс обслуживания, значения этого атрибута подлежат проверке с помощью ограничения CHECK. Точно такое же ограничение есть и в таблице «Места» (seats), в которой каждому месту в салоне конкретного типа самолета присваивается определенный класс обслуживания. Атрибут «Стоимость перелета» (amount) требует использования типа данных numeric, поскольку, как мы уже говорили ранее, денежные суммы должны записываться с определенной точностью, а гарантировать ее может только тип данных numeric. Число цифр после запятой принимается равным двум.

Оба атрибута, составляющих первичный ключ, в свою очередь, сами являются внешними ключами.

```
CREATE TABLE ticket_flights
( ticket_no       char( 13 ) NOT NULL,       -- Номер билета
  flight_id       integer NOT NULL,          -- Идентификатор рейса
  fare_conditions varchar( 10 ) NOT NULL,    -- Класс обслуживания
  amount          numeric( 10, 2 ) NOT NULL, -- Стоимость перелета
  CHECK ( amount >= 0 ),
  CHECK ( fare_conditions IN ( 'Economy', 'Comfort', 'Business' ) ),
  PRIMARY KEY ( ticket_no, flight_id ),
  FOREIGN KEY ( flight_id )
      REFERENCES flights ( flight_id ),
  FOREIGN KEY ( ticket_no )
      REFERENCES tickets ( ticket_no )
);
```

Последняя таблица нашей базы — это **«Посадочные талоны» (boarding_passes)**. Все атрибуты, представленные в ней, за исключением атрибута «Номер посадочного талона» (boarding_no), вам уже известны из других таблиц. А номер посадочного талона — это просто целое число, порядковый номер пассажира при регистрации билетов на конкретный рейс, поэтому тип данных выбирается integer.

Обратите внимание, что эта таблица имеет связь с таблицей «Перелеты» (ticket_flights) типа 1:1. Это объясняется тем, что пассажир, купивший билет на конкретный рейс, при регистрации получает только один посадочный талон. Конечно, если пассажир на регистрацию не явился, он не получает талона. Поэтому число строк в таблице «Посадочные талоны» может в общем случае оказаться меньше числа строк в таблице «Перелеты». Логично ожидать, что первичные ключи у этих двух таблиц будут одинаковыми: они включают атрибуты «Номер билета» (ticket_no) и «Идентификатор рейса» (flight_id). Поскольку таблица «Перелеты» все же является главной в этой связке таблиц, то в таблице «Посадочные талоны» создается внешний ключ, ссылающийся на нее. А поскольку тип связи между таблицами — 1:1, то внешний ключ совпадает с первичным ключом.

Известно, что номер конкретного места в самолете пассажир получает при регистрации билета, а не при его бронировании, поэтому атрибут «Номер места» (seat_no) находится в таблице «Посадочные талоны», а не в таблице «Перелеты». Нельзя допустить, чтобы на одно место в салоне были направлены два и более пассажиров, поэтому создается уникальный ключ с атрибутами «Идентификатор рейса» (flight_id) и «Номер места» (seat_no). Еще один уникальный ключ призван гарантировать несовпадение номеров посадочных талонов на данном рейсе, он включает атрибуты «Идентификатор рейса» (flight_id) и «Номер посадочного талона» (boarding_no).

```
CREATE TABLE boarding_passes
( ticket_no   char( 13 ) NOT NULL,   -- Номер билета
  flight_id   integer NOT NULL,      -- Идентификатор рейса
  boarding_no integer NOT NULL,      -- Номер посадочного талона
  seat_no     varchar( 4 ) NOT NULL, -- Номер места
  PRIMARY KEY ( ticket_no, flight_id ),
  UNIQUE ( flight_id, boarding_no ),
  UNIQUE ( flight_id, seat_no ),
  FOREIGN KEY ( ticket_no, flight_id )
      REFERENCES ticket_flights ( ticket_no, flight_id )
);
```

Вы можете, как и раньше, посмотреть описание таблицы:

```
\d boarding_passes
```

В процессе создания таблиц между ними образовывались связи за счет внешних ключей. Эти связи в описании таблицы можно увидеть, образно говоря, с двух сторон: таблицы, на которые ссылается данная таблица, указываются во фразе «Ограничения внешнего ключа», а таблицы, которые ссылаются на данную таблицу, указываются во фразе «Ссылки извне». Например:

```
\d tickets
```

```
...
Ограничения внешнего ключа:
    "tickets_book_ref_fkey" FOREIGN KEY (book_ref)
         REFERENCES bookings(book_ref)
Ссылки извне:
    TABLE "ticket_flights"
         CONSTRAINT "ticket_flights_ticket_no_fkey"
            FOREIGN KEY (ticket_no)
            REFERENCES tickets(ticket_no)
```

Наше рассмотрение команд для определения данных было бы неполным без такой важной команды, как DROP TABLE. Поскольку у вас есть файл demo_small.sql, то воссоздать таблицы базы данных будет совсем нетрудно, поэтому вы можете смело выполнять команды удаления таблиц. Давайте сначала попытаемся удалить таблицу aircrafts:

```
DROP TABLE aircrafts;
```

Казалось бы, не должно быть никаких проблем, но в результате СУБД выдает сообщение об ошибке:

```
ОШИБКА: удалить объект таблица aircrafts нельзя, так как от него зависят
 ↪  другие объекты
ПОДРОБНОСТИ: ограничение flights_aircraft_code_fkey в отношении таблица
 ↪  flights зависит от объекта таблица aircrafts
            ограничение seats_aircraft_code_fkey в отношении таблица
 ↪  seats зависит от объекта таблица aircrafts
ПОДСКАЗКА: Для удаления зависимых объектов используйте DROP ... CASCADE.
```

Дело в том, что таблица «Самолеты» (aircrafts) является ссылочной для таблиц «Рейсы» (flights) и «Места» (seats), что и отражено в сообщении. Выполнив команду

```
\d flights
```

мы увидим внешний ключ, ссылающийся на таблицу «Самолеты» (aircrafts). В сообщении содержится также и подсказка, рекомендующая в команду DROP TABLE добавить фразу CASCADE, означающую каскадное удаление зависимых объектов. Давайте так и сделаем:

```
DROP TABLE aircrafts CASCADE;
```

Теперь удаление таблицы прошло успешно, при этом из таблиц «Рейсы» (flights) и «Места» (seats) были удалены внешние ключи, ссылающиеся на удаленную таблицу aircrafts. Вот это сообщение:

```
ЗАМЕЧАНИЕ: удаление распространяется на еще 2 объекта
ПОДРОБНОСТИ: удаление распространяется на объект ограничение
 ↪  flights_aircraft_code_fkey в отношении таблица flights удаление
 ↪  распространяется на объект ограничение seats_aircraft_code_fkey в
 ↪  отношении таблица seats
DROP TABLE
```

Теперь внешних ключей, ссылающихся на таблицу aircrafts в таблицах flights и seats нет. Можно проверить это с помощью команд

```
\d flights
\d seats
```

А что если выполнить команду для удаления той же самой таблицы повторно?

```
DROP TABLE aircrafts CASCADE;
```

Ничего непоправимого не случится, просто СУБД выдаст сообщение об ошибке:

```
ОШИБКА: таблица "aircrafts" не существует
```

Однако бывают ситуации, когда заранее известно, что возможна попытка удаления несуществующей таблицы. В таких случаях обычно стараются избежать ненужных сообщений об ошибке отсутствия таблицы. Делается это путем добавления в команду DROP TABLE фразы IF EXISTS. Например:

```
DROP TABLE IF EXISTS aircrafts CASCADE;
```

При использовании этой фразы в случае наличия интересующей нас таблицы выполняется ее удаление, в случае же ее отсутствия выводится замечание, а не ошибка, а также сообщение об успешном выполнении команды удаления таблицы:

```
ЗАМЕЧАНИЕ: таблица "aircrafts" не существует, пропускается
DROP TABLE
```

## 5.3 Модификация таблиц

Модифицировать таблицы приходится по различным причинам. Например, при необходимости добавить к какому-нибудь атрибуту ограничение DEFAULT, т. е. значение «по умолчанию». Конечно, если в таблицах еще нет данных, то их можно просто пересоздать, внеся изменения в их определения. Но если таблицы содержат большое количество строк, то пересоздать их не всегда возможно, в этом случае на помощь приходит команда ALTER TABLE.

Эта команда очень многообразна и логична. Она предусматривает, наверное, все ситуации, которые могут возникнуть в реальной работе. Например, может возникнуть необходимость добавить новый столбец в таблицу — команда ALTER TABLE имеет для этого фразу ADD COLUMN. Возможна и обратная ситуация, когда нужно удалить столбец из таблицы — для этого есть фраза DROP COLUMN. Если нужно добавить ограничение, то помогут фразы ADD CHECK и ADD CONSTRAINT. Если потребовался внешний ключ, то можно добавить и его.

В качестве объектов для экспериментов будем использовать таблицы базы данных «Авиаперевозки».

Предположим, что нам понадобилось иметь в базе данных сведения о крейсерской скорости полета всех моделей самолетов, которые эксплуатируются в нашей авиакомпании. Следовательно, необходимо добавить столбец в таблицу «Самолеты» (aircrafts). Дадим ему имя speed (наверное, можно предложить и более длинное имя — cruise_speed). Тип данных для этого столбца выберем integer, добавим ограничение NOT NULL. Наложим и ограничение на минимальное значение крейсерской скорости, выраженное в километрах в час: CHECK( speed >= 300 ). В результате сформируем такую команду для добавления столбца:

```
ALTER TABLE airports
    ADD COLUMN speed integer NOT NULL CHECK( speed >= 300 );
```

При попытке выполнить эту команду СУБД выдает сообщение об ошибке:

```
ОШИБКА: столбец "speed" содержит значения NULL
```

Как понимать это сообщение: кто виноват и что делать? Дело в том, что в таблице «Самолеты» (aircrafts) уже есть строки. Однако во время добавления тех строк столбец speed в таблице не присутствовал, поэтому при его добавлении сейчас значение данного атрибута в этих строках будет отсутствовать, т. е. будет равно NULL. А мы наложили ограничение NOT NULL, следовательно, ранее добавленные строки не отвечают новому ограничению. Как же можно выйти из этой ситуации? Один из вариантов такой: сначала добавить столбец, не накладывая на его значения никаких ограничений, затем ввести значения нового атрибута в уже существующие строки, причем, эти значения должны удовлетворять тем ограничениям, которые мы собираемся наложить. После этого накладываем все необходимые ограничения. Получаем такую группу команд:

```
ALTER TABLE aircrafts ADD COLUMN speed integer;

UPDATE aircrafts SET speed = 807 WHERE aircraft_code = '733';
UPDATE aircrafts SET speed = 851 WHERE aircraft_code = '763';
UPDATE aircrafts SET speed = 905 WHERE aircraft_code = '773';
UPDATE aircrafts SET speed = 840
   WHERE aircraft_code IN ( '319', '320', '321' );

UPDATE aircrafts SET speed = 786 WHERE aircraft_code = 'CR2';
UPDATE aircrafts SET speed = 341 WHERE aircraft_code = 'CN1';
UPDATE aircrafts SET speed = 830 WHERE aircraft_code = 'SU9';

SELECT * FROM aircrafts;
ALTER TABLE aircrafts ALTER COLUMN speed SET NOT NULL;
ALTER TABLE aircrafts ADD CHECK( speed >= 300 );
```

Проверьте, как изменилось определение таблицы, с помощью команды

```
\d aircrafts
```

Конечно, если необходимость наличия того или иного ограничения отпадет, его можно удалить:

```
ALTER TABLE aircrafts ALTER COLUMN speed DROP NOT NULL;
ALTER TABLE aircrafts DROP CONSTRAINT aircrafts_speed_check;
```

Обратите внимание, что для удаления ограничения CHECK нужно указать его имя, которое можно выяснить с помощью команды

```
\d aircrafts
```

Если мы решим не усложнять нашу базу данных дополнительной информацией, то можем удалить и столбец. Конечно, вовсе не обязательно предварительно удалять ограничения, наложенные на этот столбец.

```
ALTER TABLE aircrafts DROP COLUMN speed;
```

Еще одна полезная возможность команды ALTER TABLE — изменение типа данных для какого-либо столбца. Давайте изменим тип данных для атрибутов «Координаты аэропорта: долгота» (longitude) и «Координаты аэропорта: широта» (latitude) с float (double precision) на numeric(5, 2). Сделать это можно с помощью одной команды, поскольку команда ALTER TABLE поддерживает и выполнение более одного действия за один раз.

Сначала посмотрим, с какой точностью выводятся значения этих атрибутов до изменения типа данных, затем изменим тип данных для двух столбцов, опять выведем содержимое таблицы на экран и убедимся, что значения были округлены в соответствии с правилами округления.

```
SELECT * FROM airports;

ALTER TABLE airports
  ALTER COLUMN longitude SET DATA TYPE numeric( 5,2 ),
  ALTER COLUMN latitude  SET DATA TYPE numeric( 5,2 );

SELECT * FROM airports;
```

В том случае, когда один тип данных изменяется на другой тип данных в пределах одной группы, например, оба типа — числовые, то проблем обычно не возникает. В только что рассмотренном примере исходный тип данных был float (double precision), а новый — numeric(5, 2), поэтому операция замены типа прошла автоматически.

Однако если исходный и целевой типы данных относятся к разным группам, тогда потребуются некоторые дополнительные усилия с нашей стороны. В качестве примера рассмотрим следующую ситуацию. Предположим, что по результатам опытной эксплуатации базы данных «Авиаперевозки» мы пришли к выводу о том, что необходимо создать таблицу, содержащую коды и наименования классов обслуживания. Назовем ее «Классы обслуживания» (fare_conditions). В ее состав включим два столбца: «Код класса обслуживания» и «Наименование класса обслуживания». Имена столбцам присвоим с учетом принципов формирования имен аналогичных столбцов в других таблицах, например, в таблице «Аэропорты» (airports).

```
CREATE TABLE fare_conditions
( fare_conditions_code integer,
  fare_conditions_name varchar( 10 ) NOT NULL,
  PRIMARY KEY ( fare_conditions_code )
);
```

Добавим в новую таблицу необходимые данные:

```
INSERT INTO fare_conditions
  VALUES ( 1, 'Economy' ),
         ( 2, 'Business' ),
         ( 3, 'Comfort' );
```

Поскольку мы ввели в обращение числовые коды для классов обслуживания, то необходимо модифицировать определение таблицы «Места» (seats), а именно: тип данных столбца «Класс обслуживания» (fare_conditions) изменить с varchar(10) на integer. Для реализации такой задачи служит фраза USING команды ALTER TABLE. Однако такой вариант команды не сработает:

```
ALTER TABLE seats
  ALTER COLUMN fare_conditions SET DATA TYPE integer
  USING ( CASE WHEN fare_conditions = 'Economy'  THEN 1
               WHEN fare_conditions = 'Business' THEN 2
               ELSE 3
          END );
```

Для замены исходных значений на новые мы используем конструкцию CASE WHEN … THEN … ELSE … END.

Выполнить операцию не удастся, СУБД выдаст сообщение об ошибке:

```
ОШИБКА: ограничение-проверку "seats_fare_conditions_check" нарушает
 ↪  некоторая строка
```

И в самом деле, в определении таблицы есть ограничение CHECK, которое требует, чтобы значение столбца fare_conditions выбиралось из списка: «Economy», «Comfort», «Business». При замене символьных значений на числовые это ограничение будет заведомо нарушаться. Следовательно, необходимо в команду ALTER TABLE добавить операцию удаления этого ограничения. Пробуем новый вариант команды:

```
ALTER TABLE seats
    DROP CONSTRAINT seats_fare_conditions_check,
    ALTER COLUMN fare_conditions SET DATA TYPE integer
        USING ( CASE WHEN fare_conditions = 'Economy' THEN 1
                     WHEN fare_conditions = 'Business' THEN 2
                     ELSE 3 END
        );
```

Проверим результат работы с помощью команды

```
SELECT * FROM seats;
```

```
 aircraft_code | seat_no | fare_conditions
---------------+---------+-----------------
 319           | 2A      |               2
 319           | 2C      |               2
 319           | 2D      |               2
...
```

Теперь мы видим, что необходимо связать таблицы «Места» (seats) и «Классы обслуживания» (fare_conditions) по внешнему ключу. Сделаем это:

```
ALTER TABLE seats
  ADD FOREIGN KEY ( fare_conditions )
       REFERENCES fare_conditions ( fare_conditions_code );
```

Посмотрев описание таблицы «Места» (seats), увидим, что внешний ключ успешно создан.

```
\d seats
```

```
...
"seats_fare_conditions_fkey" FOREIGN KEY (fare_conditions)
     REFERENCES fare_conditions(fare_conditions_code)
```

Из теории известно, что атрибуты внешнего ключа не обязательно должны ссылаться только на одноименные атрибуты ссылочной таблицы. Сейчас мы на практике успешно проверили это утверждение. Однако для удобства сопровождения базы данных имеет смысл переименовать столбец fare_conditions в таблице «Места» (seats), т. е. дать ему имя fare_conditions_code, поскольку в этой таблице хранится именно код класса обслуживания. Давайте так и поступим:

```
ALTER TABLE seats
     RENAME COLUMN fare_conditions TO fare_conditions_code;
```

Если теперь посмотреть описание таблицы, то можно увидеть, что имя атрибута, являющегося внешним ключом, изменилось, а вот имя ограничения seats_fare_conditions_fkey осталось неизменным, хотя оно и было первоначально сформировано самой СУБД. Это шаблонное имя ограничения составляется из имени таблицы и имени первого (и единственного в данном случае) атрибута внешнего ключа.

```
"seats_fare_conditions_fkey" FOREIGN KEY (fare_conditions_code)
 ↪  REFERENCES fare_conditions(fare_conditions_code)
```

Давайте переименуем это ограничение, чтобы поддержать соблюдение правила именования ограничений:

```
ALTER TABLE seats
    RENAME CONSTRAINT seats_fare_conditions_fkey
    TO seats_fare_conditions_code_fkey;
```

Как всегда, проверим, что получилось:

```
\d seats
```

И в заключение этого параграфа вернемся к таблице «Классы обслуживания» (fare_conditions). Мы предусмотрели в ней первичный ключ, но ведь значения атрибута «Наименование класса обслуживания» (fare_conditions_name) также должны быть уникальными, дублирование значений не допускается. Давайте добавим ограничение уникальности по этому столбцу:

```
ALTER TABLE fare_conditions ADD UNIQUE ( fare_conditions_name );
```

И как всегда, на всякий случай проверим, что получилось:

```
\d fare_conditions
```

## 5.4 Представления

При работе с базами данных зачастую приходится многократно выполнять одни и те же запросы, которые могут быть весьма сложными и требовать обращения к нескольким таблицам. Чтобы избежать необходимости многократного формирования таких запросов, можно использовать так называемые представления (views). Если речь идет о выборке данных, то представления практически неотличимы от таблиц с точки зрения обращения к ним в командах SELECT.

Упрощенный синтаксис команды CREATE VIEW, предназначенной для создания представлений, таков:

```
CREATE VIEW name [ ( column_name [, ...] ) ]
    AS query;
```

В этой команде обязательными элементами являются имя представления и запрос к базе данных, который и формирует выборку из нее. Если список имен столбцов не приведен, тогда их имена «вычисляются» (формируются) на основании текста запроса.

Давайте создадим простое представление. В главе 3 мы решали задачу: подсчитать количество мест в салонах для всех моделей самолетов с учетом класса обслуживания (бизнес-класс и экономический класс). Запрос был таким:

```
SELECT aircraft_code, fare_conditions, count( * )
    FROM seats
    GROUP BY aircraft_code, fare_conditions
    ORDER BY aircraft_code, fare_conditions;
```

На его основе создадим представление и дадим ему имя, отражающее суть этого представления.

```
CREATE VIEW seats_by_fare_cond AS
    SELECT aircraft_code, fare_conditions, count( * )
        FROM seats
        GROUP BY aircraft_code, fare_conditions
        ORDER BY aircraft_code, fare_conditions;
```

Теперь мы можем вместо написания сложного первоначального запроса обращаться непосредственно к представлению, как будто это обычная таблица.

```
SELECT * FROM seats_by_fare_cond;
```

В отличие от таблиц, представления не содержат данных. Данные выбираются из таблиц, на основе которых представление создано, при каждом обращении к нему в команде SELECT.

СУБД PostgreSQL предлагает свое расширение команды CREATE VIEW, а именно: фразу OR REPLACE. Если представление уже существует, то можно его не удалять, а просто заменить новой версией. Однако нужно помнить о том, что при создании новой версии представления (без явного удаления старой с помощью команды DROP VIEW) должны оставаться неизменными имена столбцов представления. Если же вы хотите изменить имя хотя бы одного столбца, то сначала нужно удалить представление с помощью команды DROP VIEW, а уже затем создать его заново. Имена столбцов можно явно указать в команде, но если они не указаны, то СУБД сама «вычислит» эти имена. В только что созданном нами представлении третий столбец получит имя count. Если мы захотим изменить это имя, то возможны два способа: первый заключается в том, чтобы создать псевдоним для этого столбца с помощью ключевого слова AS, а второй — в использовании списка имен столбцов в начале команды CREATE VIEW.

Попробуем воспользоваться первым способом (обратите внимание на добавление фразы OR REPLACE и ключевого слова AS после вызова функции count):

```
CREATE OR REPLACE VIEW seats_by_fare_cond AS
    SELECT a.model, s.aircraft_code, s.fare_conditions,
           count( * ) AS num_seats
        FROM seats
        GROUP BY aircraft_code, fare_conditions
        ORDER BY aircraft_code, fare_conditions;
```

Однако СУБД выдаст сообщение об ошибке:

```
ОШИБКА: изменить имя столбца "count" на "num_seats" в представлении
 ↪  нельзя
```

В чем дело? А дело в том, что при первоначальном создании этого представления третий столбец уже получил имя count (такое имя ему дала СУБД). Поэтому если мы хотим воспользоваться вариантом команды с фразой OR REPLACE, мы не должны изменять названия столбцов ни путем указания псевдонимов, ни с помощью списка имен столбцов, приводимого в начале команды. Так что если мы все же захотим изменить имя столбца в представлении, нам придется сначала удалить это представление, а затем создать его заново.

```
DROP VIEW seats_by_fare_cond;

CREATE OR REPLACE VIEW seats_by_fare_cond AS
    SELECT a.model, s.aircraft_code, s.fare_conditions,
           count( * ) AS num_seats
        FROM seats
        GROUP BY aircraft_code, fare_conditions
        ORDER BY aircraft_code, fare_conditions;
```

А вот и второй способ задания имен столбцов в представлении — с помощью списка их имен, заключенного в скобки:

```
DROP VIEW seats_by_fare_cond;

CREATE OR REPLACE VIEW seats_by_fare_cond
    ( code, fare_cond, num_seats )
    AS
    SELECT aircraft_code, fare_conditions, count( * )
      FROM seats
      GROUP BY aircraft_code, fare_conditions
      ORDER BY aircraft_code, fare_conditions;
```

Представления позволяют облегчить развитие и модификацию базы данных, потому что они могут позволить сохранить интерфейс неизменным, но сам запрос, который лежит в основе конкретного представления, может измениться. При этом для прикладного программиста представление останется неизменным, поэтому не потребуется переделывать запросы к этому представлению в прикладной программе.

В базе данных «Авиаперевозки» создано представление «Рейсы» (flights_v), сконструированное на основе таблицы «Рейсы» (flights), но содержащее дополнительную информацию, а именно:

- подробные сведения об аэропорте вылета (departure_airport, departure_airport_name, departure_city);
- подробные сведения об аэропорте прибытия (arrival_airport, arrival_airport_name, arrival_city);
- местное время вылета, как плановое, так и фактическое (scheduled_departure_local, actual_departure_local);
- местное время прибытия, как плановое, так и фактическое (scheduled_arrival_local, actual_arrival_local);

– продолжительность полета, как плановая, так и фактическая (scheduled_duration, actual_duration).

Мы только опишем все столбцы представления, а SQL-команду для его создания приведем в главе 6.

| Описание атрибута | Имя атрибута | Тип PostgreSQL |
|---|---|---|
| Идентификатор рейса | flight_id | integer |
| Номер рейса | flight_no | char(6) |
| Время вылета по расписанию | scheduled_departure | timestamptz |
| Время вылета по расписанию, местное время в пункте отправления | scheduled_departure_local | timestamp |
| Время прилета по расписанию | scheduled_arrival | timestamptz |
| Время прилета по расписанию, местное время в пункте прибытия | scheduled_arrival_local | timestamp |
| Планируемая продолжительность полета | scheduled_duration | interval |
| Код аэропорта отправления | departure_airport | char(3) |
| Название аэропорта отправления | departure_airport_name | text |
| Город отправления | departure_city | text |
| Код аэропорта прибытия | arrival_airport | char(3) |
| Название аэропорта прибытия | arrival_airport_name | text |
| Город прибытия | arrival_city | text |
| Статус рейса | status | varchar(20) |
| Код самолета, IATA | aircraft_code | char(3) |
| Фактическое время вылета | actual_departure | timestamptz |
| Фактическое время вылета, местное время в пункте отправления | actual_departure_local | timestamp |
| Фактическое время прилета | actual_arrival | timestamptz |
| Фактическое время прилета, местное время в пункте прибытия | actual_arrival_local | timestamp |
| Фактическая продолжительность полета | actual_duration | interval |

Известно, что в сфере железнодорожных пассажирских перевозок время в расписании движения поездов и в билетах указывается московское. А в пассажирских авиаперевозках, напротив, время в билетах указывается местное. Это касается и времени вылета и времени прилета. Если пункты отправления и назначения находятся в различных часовых поясах, то время вылета будет привязано к одному часовому поясу, а время прилета — к другому.

Поэтому в нашем представлении «Рейсы» (flights_v) предусмотрены четыре столбца, отображающие местное время: два из них относятся к пункту отправления — scheduled_departure_local и actual_departure_local, а два других относятся к пункту прибытия — scheduled_arrival_local и actual_arrival_local. В качестве типа данных для этих четырех столбцов выбран тип timestamp without time zone (сокращенно — просто timestamp), а не timestamp with time zone (timestamptz). Причина в том, что при выборе timestamptz время автоматически преобразовывалось бы при выводе данных к текущему часовому поясу, установленному на компьютере пользователя, а нам нужно сохранить его значения такими, какими они являются в пункте отправления и пункте назначения.

Для перевода значения типа timestamptz в значение типа timestamp служит конструкция AT TIME ZONE, подробно рассмотренная в разделе документации 9.9 «Операторы и функции даты/времени». Также существует и эквивалентная функция timezone, которая и используется здесь для пересчета московского времени в местное.

Если вы испытываете затруднения с пониманием операций преобразования значений типа timestamptz в значения типа timestamp, рекомендуем вам обратиться к разделу документации 8.5.1.3 «Даты и время».

Посмотреть описание представления в базе данных можно с помощью команды

```
\d flights_v
```

В представлении «Рейсы» (flights_v) много столбцов, поэтому при выводе информации из него в виде таблицы каждая строка на экране будет сворачиваться «змейкой», что не очень наглядно. Утилита psql предлагает альтернативный — расширенный — способ вывода информации, который включается с помощью команды

```
\x
```

Для возвращения к табличному формату вывода нужно выполнить эту же команду еще раз.

Включив расширенный вывод, выполните команду для выборки данных из представления «Рейсы» (flights_v):

```
SELECT * FROM flights_v;
```

Будет получен вот такой результат:

```
-[ RECORD 1 ]--------------+-------------------------
 flight_id                 | 1
 flight_no                 | PG0405
 scheduled_departure       | 2016-09-13 13:35:00+08
 scheduled_departure_local | 2016-09-13 08:35:00
 scheduled_arrival         | 2016-09-13 14:30:00+08
 scheduled_arrival_local   | 2016-09-13 09:30:00
 scheduled_duration        | 00:55:00
 departure_airport         | DME
 departure_airport_name    | Домодедово
 departure_city            | Москва
 arrival_airport           | LED
 arrival_airport_name      | Пулково
 arrival_city              | Санкт-Петербург
 status                    | Arrived
 aircraft_code             | 321
 actual_departure          | 2016-09-13 13:44:00+08
 actual_departure_local    | 2016-09-13 08:44:00
 actual_arrival            | 2016-09-13 14:39:00+08
 actual_arrival_local      | 2016-09-13 09:39:00
 actual_duration           | 00:55:00...
...
```

Бывают ситуации, когда заранее известно, что возможна попытка удаления несуществующего представления. В таких случаях обычно стараются избежать ненужных сообщений об ошибке отсутствия представления. Делается это путем добавления в команду DROP VIEW фразы IF EXISTS. Например:

```
DROP VIEW IF EXISTS flights_v;
```

Как мы уже говорили ранее, представление является фактически сохраненным запросом к базе данных. Этот запрос получает имя, которым можно впоследствии воспользоваться в предложении FROM команды SELECT для получения результатов этого запроса.

PostgreSQL предлагает свое расширение — так называемое материализованное представление. Упрощенный синтаксис команды CREATE MATERIALIZED VIEW, предназначенной для создания материализованных представлений, таков:

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] table_name
    [ (column_name [, ...] ) ]
    AS query
    [ WITH [ NO ] DATA ];
```

Материализованное представление заполняется данными в момент выполнения команды для его создания, если только в команде не было фразы WITH NO DATA. Если же она была включена в команду, тогда в момент своего создания представление данными не заполняется, а для заполнения его данными нужно использовать команду REFRESH MATERIALIZED VIEW.

Материализованное представление очень похоже на обычную таблицу. Однако оно отличается от таблицы тем, что не только сохраняет данные, но также запоминает запрос, с помощью которого эти данные были собраны.

В нашей учебной базе данных «Авиаперевозки» имеется материализованное представление — «Маршруты» (routes). Как вы могли заметить, таблица «Рейсы» (flights) содержит избыточность: для одного и того же номера рейса, отправляющегося в различные дни, повторяются коды аэропортов отправления и назначения, а также код самолета. Таким образом, из этой таблицы можно извлечь информацию о маршруте, т. е. номер рейса, аэропорты отправления и назначения. Эта информация не зависит от конкретной даты вылета.

Опишем все столбцы представления, а SQL-команду для его создания приведем в главе 6.

| Описание атрибута | Имя атрибута | Тип PostgreSQL |
|---|---|---|
| Номер рейса | flight_no | char(6) |
| Код аэропорта отправления | departure_airport | char(3) |
| Название аэропорта отправления | departure_airport_name | text |
| Город отправления | departure_city | text |
| Код аэропорта прибытия | arrival_airport | char(3) |
| Название аэропорта прибытия | arrival_airport_name | text |
| Город прибытия | arrival_city | text |
| Код самолета, IATA | aircraft_code | char(3) |
| Продолжительность полета | duration | interval |
| Дни недели, когда выполняются рейсы | days_of_week | integer[] |

Обратите внимание на тип данных последнего столбца — «Дни недели, когда выполняются рейсы». Это массив целых чисел.

Если впоследствии вам потребуется обновить данные в материализованном представлении, то выполните команду

```
REFRESH MATERIALIZED VIEW routes;
```

Кончено, как и любой другой объект базы данных, материализованное представление можно удалить.

```
DROP MATERIALIZED VIEW routes;
```

Подводя итог параграфа, назовем положительные стороны использования представлений.

1. Упрощение разграничения полномочий пользователей на доступ к хранимым данным.

   Разным типам пользователей могут требоваться различные данные, хранящиеся в одних и тех же таблицах. Это касается как столбцов, так и строк таблиц. Создание различных представлений для разных пользователей избавляет от необходимости создавать дополнительные таблицы, дублируя данные, и упрощает организацию системы управления доступом к данным.

2. Упрощение запросов к базе данных.

   Запросы к базе данных могут включать несколько таблиц и быть весьма сложными и громоздкими, при этом такие запросы могут выполняться часто. Использование представлений позволяет скрыть эти сложности от прикладного программиста и сделать запросы более простыми и наглядными.

3. Снижение зависимости прикладных программ от изменений структуры таблиц базы данных.

   В процессе развития информационной системы структура таблиц базы данных может изменяться. Столбцы представления, т. е. их имена, типы данных и порядок следования, — это, образно говоря, интерфейс к запросу, который реализуется данным представлением. Если этот интерфейс остается неизменным, то SQL-запросы, в которых используется данное представление, корректировать не потребуется. Нужно будет лишь в ответ на изменение структуры базовых таблиц, на основе которых представление сконструировано, соответствующим образом перестроить запрос, выполняемый данным представлением.

4. Снижение времени выполнения сложных запросов за счет использования материализованных представлений.

   В материализованных представлениях можно сохранять результаты выполнения запросов, которые формируются длительное время, но при этом допускают их формирование заранее, а не обязательно в момент возникновения потребности в результатах этого запроса. Если, например, какой-нибудь сводный отчет формируется длительное время, а запросы к отчету будут неоднократными, то может оказаться целесообразным сформировать его заранее и сохранить в материализованном представлении.

Тем не менее, нужно учитывать, что применимость материализованных представлений весьма ограничена. Не следует заменять ими все сложные запросы. Одним из их недостатков является то, что их необходимо своевременно обновлять с помощью команды REFRESH, чтобы они содержали актуальные данные.

## 5.5 Схемы базы данных

Схема — это логический фрагмент базы данных, в котором могут содержаться различные объекты: таблицы, представления, индексы и др. В базе данных обязательно есть хотя бы одна схема. При создании базы данных в ней автоматически создается схема с именем public. Когда мы с вами создавали таблицы в базе данных edu, они создавались именно в этой схеме.

В каждой базе данных может содержаться более одной схемы. Их имена должны быть уникальными в пределах конкретной базы данных. Имена объектов базы данных (таблиц, представлений, последовательностей и др.) должны быть уникальными в пределах конкретной схемы, но в разных схемах имена объектов могут повторяться. Таким образом, можно сказать, что схема образует так называемое *пространство имен*.

Посмотреть список схем в базе данных можно так:

```
\dn
```

```
Список схем
   Имя    | Владелец
----------+----------
 bookings | postgres
 public   | postgres
(2 строки)
```

В учебной базе данных demo есть схема bookings. Все таблицы созданы именно в этой схеме. Для организации доступа к ней вы уже выполняли команду

```
SET search_path = bookings;
```

Теперь объясним подробнее, что эта команда делает.

Если в базе данных создано более одной схемы, то доступ к объектам, содержащимся в конкретной схеме, можно организовать разными способами. Первый заключается в том, чтобы имена объектов предварять именем схемы. Например, для обращения к таблице aircrafts нужно сделать так:

```
SELECT * FROM bookings.aircrafts;
```

Однако такой способ не очень удобен. Другой способ заключается в том, чтобы одну из схем сделать *текущей*. Среди параметров времени исполнения, которые предусмотрены в конфигурации сервера PostgreSQL, есть параметр search_path. Его значение по умолчанию можно изменить в конфигурационном файле postgresql.conf. Он содержит имена схем, которые PostgreSQL просматривает при поиске конкретного объекта базы данных, когда имя схемы в команде не указано. Посмотреть значение этого параметра можно с помощью команды SHOW:

```
SHOW search_path;
```

```
   search_path
-----------------
 "$user", public
(1 строка)
```

Схема "$user" присутствует в этом параметре на тот случай, если будут созданы схемы с именами, совпадающими с именами пользователей. Тогда могут упроститься некоторые операции с базой данных. Однако в базе данных demo нет таких схем, поэтому первый элемент параметра search_path фактически не участвует в работе, в результате все обращения к объектам базы данных без указания имени схемы будут адресоваться схеме public.

Чтобы изменить порядок просмотра схем при поиске объектов в базе данных, нужно воспользоваться командой SET. При этом первой в списке схем следует указать именно ту, которую СУБД должна просматривать первой. Эта схема и станет текущей. Конечно, такой список может состоять и всего из одной схемы.

Давайте выполним команду

```
SET search_path = bookings;
```

А теперь посмотрим, что получилось:

```
SHOW search_path;
```

```
 search_path
-------------
 bookings
(1 строка)
```

Да, действительно, теперь первой будет просматриваться схема bookings. А для обращения к объектам, например, таблицам, в схеме public (если бы они в ней были) нам пришлось бы указывать имя схемы public перед именами этих объектов. Если бы мы решили добавить схему public в список просматриваемых схем, то нужно было бы включить ее в команду SET:

```
SET search_path = bookings, public;
```

Узнать имя текущей схемы можно с помощью встроенной функции current_schema (обратите внимание на отсутствие скобок при вызове функции в команде SELECT).

```
SELECT current_schema;
```

```
 current_schema
----------------
 bookings
(1 строка)
```

При создании объектов базы данных, например, таблиц, необходимо учитывать следующее: если имя схемы в команде не указано, то объект будет создан в текущей схеме. Если же вы хотите создать объект в конкретной схеме, которая не является текущей, то нужно указать ее имя перед именем создаваемого объекта, разделив их точкой. Например, для создания таблицы airports в схеме my_schema следует сделать так:

```
CREATE TABLE my_schema.airports
...
```

## Контрольные вопросы и задания

1. При использовании значений по умолчанию с ключевым словом DEFAULT возможны и ситуации, когда типичным будет не конкретное значение данных, а способ его получения. Например, если мы захотим фиксировать в каждой строке таблицы «Студенты» (students) имя пользователя базы данных, добавившего эту строку в таблицу, тогда необходимо в определение таблицы добавить еще один столбец. Этот столбец по умолчанию будет получать значение, возвращаемое функцией current_user.

   ```
   CREATE TABLE students
   (  record_book numeric( 5 ) NOT NULL,
      name text NOT NULL,
      doc_ser numeric( 4 ),
      doc_num numeric( 6 ),
      who_adds_row text DEFAULT current_user, -- добавленный столбец
      PRIMARY KEY ( record_book )
   );
   ```

   Эта функция — current_user — будет вызываться не при создании таблицы, а при вставке каждой строки. При этом в команде INSERT не требуется указывать значение для столбца who_adds_row, поскольку функция current_user будет вызываться самой СУБД PostgreSQL:

   ```
   INSERT INTO students ( record_book, name, doc_ser, doc_num )
       VALUES ( 12300, 'Иванов Иван Иванович', 0402, 543281 );
   ```

   Давайте пойдем дальше и пожелаем фиксировать не только имя пользователя базы данных, добавившего строку в таблицу, но также и момент времени, когда это было сделано. Самостоятельно внесите модификацию в определение таблицы students для решения этой задачи, а затем выполните команду INSERT для проверки полученного решения.

   Если до выполнения этого упражнения вы еще не ознакомились с командой ALTER TABLE, то вместо модифицирования определения таблицы сначала удалите ее, а затем создайте заново:

   ```
   DROP TABLE students;
   CREATE TABLE students ...
   ```

2. Посмотрите, какие ограничения уже наложены на атрибуты таблицы «Успеваемость» (progress). Воспользуйтесь командой \d утилиты psql. А теперь предложите для этой таблицы ограничение уровня таблицы.

   В качестве примера рассмотрим такой вариант. Добавьте в таблицу progress еще один атрибут — «Форма проверки знаний» (test_form), который может принимать только два значения: «экзамен» или «зачет». Тогда набор допустимых значений атрибута «Оценка» (mark) будет зависеть от того, экзамен или зачет

предусмотрены по данной дисциплине. Если предусмотрен экзамен, тогда допускаются значения 3, 4, 5, если зачет — тогда 0 (не зачтено) или 1 (зачтено). Не забудьте, что значения NULL для атрибутов test_form и mark не допускаются.

Новое ограничение может быть таким:

```
ALTER TABLE progress
  ADD CHECK (
    ( test_form = 'экзамен' AND mark IN ( 3, 4, 5 ) )
    OR
    ( test_form = 'зачет' AND mark IN ( 0, 1 ) )
  );
```

Проверьте, как будет работать новое ограничение в модифицированной таблице progress. Для этого выполните команды INSERT как удовлетворяющие ограничению, так и нарушающие его.

В таблице уже было ограничение на допустимые значения атрибута mark. Как вы думаете, не будет ли оно конфликтовать с новым ограничением? Проверьте эту гипотезу. Если ограничения конфликтуют, тогда удалите старое ограничение и снова попробуйте добавить строки в таблицу.

Подумайте, какое еще ограничение уровня таблицы можно предложить для этой таблицы?

3.* В определении таблицы «Успеваемость» (progress) на атрибуты term и mark наложены как ограничения CHECK, так и ограничение NOT NULL. Возникает вопрос: не является ли ограничение NOT NULL избыточным? Ведь мы в ограничении CHECK явно указали допустимые значения. Проверьте гипотезу об избыточности ограничения NOT NULL в данном случае. Для этого модифицируйте таблицу, убрав ограничение NOT NULL, и попробуйте добавить в нее строку с отсутствующим значением атрибута term (или mark).

4. В определении таблицы «Успеваемость» (progress) для атрибута mark не только задано ограничение CHECK, но и установлено значение по умолчанию с помощью ключевого слова DEFAULT:

```
 mark numeric( 1 ) NOT NULL
   CHECK ( mark >= 3 AND mark <= 5 )
   DEFAULT 5,
```

Как вы думаете, что будет, если в ограничении DEFAULT мы «случайно» допустим ошибку, написав DEFAULT 6? На каком этапе эта ошибка будет выявлена: уже на этапе создания таблицы или только при вставке строки в нее, если в команде INSERT не указать значение для атрибута mark?

Вот эта команда может быть вам полезной для проверки гипотезы, поскольку в ней отсутствует передаваемое значение для атрибута mark:

```
INSERT INTO progress ( record_book, subject, acad_year, term )
    VALUES ( 12300, 'Физика', '2016/2017',1 );
```

5. В стандарте SQL сказано, что при наличии ограничения уникальности, включающего один или более столбцов, все же возможны повторяющиеся значения этих столбцов в разных строках, но лишь в том случае, если это значения — NULL. PostgreSQL придерживается такого же подхода.

   Модифицируйте определение таблицы «Студенты» (students), добавив ограничение уникальности по двум столбцам: doc_ser и doc_num. А затем проверьте вышеприведенное утверждение, добавив в таблицу не только строки, содержащие конкретные значения этих двух столбцов, но также и по две строки, имеющие следующие свойства:

   – одинаковые значения столбца doc_ser и NULL-значения столбца doc_num;
   – NULL-значения столбца doc_num и столбца doc_ser.

   Подобные вещи возможны, так как NULL-значения не считаются совпадающими. Это можно проверить с помощью команды

   ```
   SELECT (null = null);
   ```

   Она даст такой результат (т. е. NULL):

   ```
    ?column?
   ----------

    (1 строка)
   ```

6. Модифицируйте определения таблиц «Студенты» (students) и «Успеваемость» (progress). В таблице students в качестве первичного ключа назначьте комбинацию атрибутов doc_ser и doc_num, а в таблице progress соответствующим образом измените определение внешнего ключа.

   ```
   CREATE TABLE students
   ( record_book numeric( 5 ) NOT NULL UNIQUE,
     name text NOT NULL,
     doc_ser numeric( 4 ),
     doc_num numeric( 6 ),
     PRIMARY KEY ( doc_ser, doc_num )
   );
   ```

   Обратите внимание, что для атрибутов doc_ser и doc_num можно не указывать ограничение NOT NULL: они входят в состав первичного ключа, а в нем NULL-значения не допускаются, поэтому ограничение NOT NULL фактически подразумевается при включении атрибута в состав первичного ключа.

   ```
   CREATE TABLE progress
   ( doc_ser numeric( 4 ),
     doc_num numeric( 6 ),
     subject text NOT NULL,
     acad_year text NOT NULL,
     term numeric( 1 ) NOT NULL CHECK ( term = 1 OR term = 2 ),
     mark numeric( 1 ) NOT NULL CHECK ( mark >= 3 AND mark <= 5 )
       DEFAULT 5,
     FOREIGN KEY ( doc_ser, doc_num )
       REFERENCES students ( doc_ser, doc_num )
       ON DELETE CASCADE
   ```

```
    ON UPDATE CASCADE
);
```

Теперь и первичный, и внешний ключи — составные. Проверьте их действие, добавив несколько строк в каждую таблицу.

7.* Модифицируйте определение таблицы «Успеваемость» (progress), а если потребуется, то и определение таблицы «Студенты» (students), чтобы изучить все варианты реагирования СУБД на обновление строк в ссылочной таблице, в данном случае — students. Последовательно изменяйте определение внешнего ключа таблицы progress, испробовав варианты ON UPDATE CASCADE, ON UPDATE RESTRICT, ON UPDATE SET NULL и ON UPDATE SET DEFAULT. Для получения информативной картины введите несколько строк в обе таблицы, а затем выполняйте операцию UPDATE, подбирая значения ключевых атрибутов таким образом, чтобы вызвать ожидаемую реакцию СУБД.

Учтите, что при использовании фразы ON UPDATE SET DEFAULT необходимо, чтобы, во-первых, с помощью ключевого слова DEFAULT было установлено значение по умолчанию для атрибута внешнего ключа в ссылающейся таблице, а во-вторых, это DEFAULT-значение все равно должно присутствовать в одной из строк ссылочной таблицы. Как вы считаете, с учетом сказанного, возможно ли использование ON UPDATE SET DEFAULT в нашем случае?

Попробуйте обосновать или, наоборот, опровергнуть целесообразность использования каждой из этих политик — CASCADE, RESTRICT, SET NULL и SET DEFAULT — при выполнении операции UPDATE в реальной информационной системе, предназначенной для учета успеваемости студентов.

8. В таблице «Успеваемость» (progress) есть атрибут «Учебная дисциплина» (subject). Это текстовый атрибут. Одинаковые наименования учебных дисциплин записываются в таблицу progress многократно. Создайте еще одну таблицу — «Учебные дисциплины» (subjects), в которой будет два атрибута: «Идентификатор учебной дисциплины» (subject_id) и «Учебная дисциплина» (subject). Тип данных первого из них будет integer, а второго — text. В качестве первичного ключа будет служить subject_id, а второй атрибут будет уникальным. Введите в новую таблицу две-три строки для различных учебных дисциплин.

Модифицируйте таблицу progress, заменив атрибут subject на subject_id. Тип данных нового атрибута будет integer. Поскольку тип данных изменится, то для замены первоначальных значений, хранящихся в этом столбце, на новые придется использовать конструкцию USING (о ней говорится в тексте главы).

Добавьте в определение таблицы progress еще один внешний ключ, который будет ссылаться на таблицу subjects. В составе этого внешнего ключа будет только один атрибут — subject_id.

Мы видим, что таблица может иметь больше одного внешнего ключа. Таким образом, структура связей в реальной базе данных может оказаться весьма сложной.

Теперь введите несколько строк и в таблицу progress, учитывая ее связь с новой таблицей subjects.

9. В таблице «Студенты» (students) есть текстовый атрибут name, на который наложено ограничение NOT NULL. Как вы думаете, что будет, если при вводе новой строки в эту таблицу дать атрибуту name в качестве значения пустую строку? Например:

```
INSERT INTO students ( record_book, name, doc_ser, doc_num )
    VALUES ( 12300, '', 0402, 543281 );
```

Наверное, проектируя эту таблицу, мы хотели бы все же, чтобы пустые строки в качестве значения атрибута name не проходили в базу данных? Какое решение вы можете предложить? Видимо, нужно добавить ограничение CHECK для столбца name. Если вы еще не изучили команду ALTER TABLE, то удалите таблицу students и создайте ее заново с учетом нового ограничения, а если с командой ALTER TABLE вы уже познакомились, то сделайте так:

```
ALTER TABLE students ADD CHECK ( name <> '' );
```

Добавив ограничение, попробуйте теперь вставить в таблицу students сроку (row), в которой значение атрибута name было бы пустой строкой (string).

Давайте продолжим эксперименты и предложим в качестве значения атрибута name строку, содержащую сначала один пробел, а потом — два пробела.

```
INSERT INTO students VALUES ( 12346, ' ', 0406, 112233 );
INSERT INTO students VALUES ( 12347, ' ', 0407, 112234 );
```

Для того чтобы «увидеть» эти пробелы в выборке, сделаем так:

```
SELECT *, length( name ) FROM students;
```

Оказывается, эти невидимые значения имеют ненулевую длину. Что делать, чтобы не допустить таких значений-невидимок? Один из способов: возложить проверку таких ситуаций на прикладную программу. А что можно сделать на уровне определения таблицы students? Какое ограничение нужно предложить? В разделе 9.4 «Строковые функции и операторы» есть функция trim(). Попробуйте воспользоваться ею. Если вы еще не изучили команду ALTER TABLE, то удалите таблицу students и создайте ее заново с учетом нового ограничения, а если с командой ALTER TABLE вы уже познакомились, то сделайте так:

```
ALTER TABLE students ADD CHECK (...);
```

Посмотрите и таблицу «Успеваемость» (progress) на предмет подобных слабых мест.

10. В таблице «Студенты» (students) атрибут «Серия документа, удостоверяющего личность» (doc_ser) имеет числовой тип, однако в сериях таких документов могут встречаться лидирующие нули, которые в числовых столбцах не сохраняются. Например, при записи значения серии «0402» первый ноль не сохранится в таблице.

Модифицируйте таблицу students, заменив числовой тип данных на символьный, например, character. Как вы думаете, эта операция пройдет без затруднений или они все же возможны? Проверьте ваши предположения, выполнив модификацию.

11.* В таблице «Рейсы» (flights) есть ограничение, которое регулирует соотношения значений фактического времени вылета и фактического времени прилета. Как вы думаете, не является ли выражение actual_arrival IS NOT NULL во второй части условного оператора OR избыточным?

```
CREATE TABLE flights
( ...
    CHECK ( actual_arrival IS NULL OR
            ( actual_departure IS NOT NULL AND
              actual_arrival IS NOT NULL AND
              actual_arrival > actual_departure
            )
    ),
...
```

Проверьте ваши предположения на практике. Для этого сначала удалите существующее ограничение с помощью команды

```
ALTER TABLE flights DROP CONSTRAINT имя_ограничения;
```

Как определить имя этого ограничения? С помощью команды

```
\d flights
```

получите описание таблицы flights, а в нем есть названия всех ограничений.

Затем создайте это же ограничение, но в модифицированном виде:

```
ALTER TABLE flights
  ADD CHECK ( actual_arrival IS NULL OR
    ( actual_departure IS NOT NULL AND
      actual_arrival > actual_departure
    )
  );
```

Попробуйте добавить в таблицу flights две-три строки, подбирая такие значения атрибутов actual_departure и actual_arrival, чтобы проверить все возможные исходы этих проверок. Конечно, вместо добавления новых строк можно модифицировать одну и ту же строку с помощью команды UPDATE.

12. Команда ALTER TABLE позволяет переименовать таблицу. Например:

```
ALTER TABLE table_name RENAME TO new_table_name;
```

Поскольку в командах создания таблиц базы данных «Авиаперевозки» мы не указывали имена ограничений для первичных и внешних ключей, то их имена были сформированы автоматически самой СУБД. Как вы думаете, получили ли эти ограничения новые имена после переименования таблицы?

Проверьте ваши предположения, выполнив такую операцию с одной из таблиц базы данных «Авиаперевозки», имеющих внешние ключи.

13. Представление «Рейсы» (flights_v) и материализованное представление «Маршруты» (routes) построены на основе таблиц «Рейсы» (flights) и «Аэропорты» (airports). Логично предположить, что при каскадном удалении, например, таблицы «Аэропорты», представление «Рейсы» будет также удалено, поскольку при удалении базовой таблицы этому представлению просто неоткуда будет брать данные. А что вы можете предположить насчет материализованного представления «Маршруты»: будет ли оно также удалено или нет? Ведь оно уже *содержит* данные, в отличие от обычного представления. Так ли, условно говоря, сильна его связь с таблицами, на основе которых оно сконструировано?

    Проведите необходимые эксперименты, начав с команды

    ```
    DROP TABLE airports;
    ```

    Если вам потребуется восстановить все объекты базы данных, то вы всегда сможете воспользоваться файлом demo_small.sql и просто повторить процедуру развертывания учебной базы данных, которая описана в главе 2. Поэтому смело экспериментируйте с таблицами и представлениями.

14. Представления (views) могут быть обновляемыми. Это значит, что можно с помощью команд INSERT, UPDATE и DELETE, применяемых к представлению, внести изменения в таблицу, лежащую в основе этого представления.

    Самостоятельно ознакомьтесь с этим вопросом с помощью документации (см. описание команды CREATE VIEW) и, создав простое представление над одной из таблиц базы данных «Авиаперевозки», выполните несколько команд с целью внесения изменений в эту таблицу.

15. Определение таблицы можно изменить с помощью команды ALTER TABLE. Аналогичные команды существуют и для изменения представлений и материализованных представлений: ALTER VIEW и ALTER MATERIALIZED VIEW. Самостоятельно ознакомьтесь с их возможностями с помощью документации.

16. Как вы думаете, при изменении данных в таблицах, на основе которых сконструировано материализованное представление, содержимое этого представления тоже синхронно изменяется или нет?

    Если содержимое материализованного представления изменяется синхронно с базовыми таблицами, то продемонстрируйте это. Если же оно остается неизменным, то покажите, как его синхронизировать с базовыми таблицами.

17. Представления могут быть, условно говоря, *вертикальными* и *горизонтальными*. При создании вертикального представления в список его столбцов включается лишь часть столбцов базовой таблицы (таблиц). Например:

    ```
    CREATE VIEW airports_names AS
      SELECT airport_code, airport_name, city
      FROM airports;

    SELECT * FROM airports_names;
    ```

    В горизонтальное представление включаются не все строки базовой таблицы (таблиц), а производится их отбор с помощью фраз WHERE или HAVING. Например:

```
CREATE VIEW siberian_airports AS
  SELECT * FROM airports
  WHERE city = 'Новосибирск' OR city = 'Кемерово';

SELECT * FROM siberian_airports;
```

Конечно, вполне возможен и смешанный вариант, когда ограничивается как список столбцов, так и множество строк при создании представления.

Подумайте, какие представления было бы целесообразно создать для нашей базы данных «Авиаперевозки». Необходимо учесть наличие различных групп пользователей, например: пилоты, диспетчеры, пассажиры, кассиры. Создайте представления и проверьте их в работе.

18.* Предположим, что нам понадобилось иметь в базе данных сведения о технических характеристиках самолетов, эксплуатируемых в авиакомпании. Пусть это будут такие сведения, как число членов экипажа (пилоты), тип двигателей и их количество. Следовательно, необходимо добавить столбец в таблицу «Самолеты» (aircrafts). Дадим ему имя specifications, а в качестве типа данных выберем jsonb. Если впоследствии потребуется добавить и другие характеристики, то мы сможем это сделать, не модифицируя определение таблицы.

```
ALTER TABLE aircrafts ADD COLUMN specifications jsonb;
```

```
ALTER TABLE
```

Добавим сведения для модели самолета Airbus A320-200:

```
UPDATE aircrafts
  SET specifications =
      '{ "crew": 2,
         "engines": { "type": "IAE V2500", "num": 2 }
       }'::jsonb
  WHERE aircraft_code = '320';
```

```
UPDATE 1
```

Посмотрим, что получилось:

```
SELECT model, specifications
  FROM aircrafts
  WHERE aircraft_code = '320';
```

```
     model       |               specifications
-----------------+-----------------------------------------------
Airbus A320-200  | {"crew": 2, "engines": {"num": 2, "type":
                    "IAE V2500"}}
(1 строка)
```

Можно посмотреть только сведения о двигателях:

```
SELECT model, specifications->'engines' AS engines
  FROM aircrafts
  WHERE aircraft_code = '320';
```

```
     model       |           engines
-----------------+---------------------------------
 Airbus A320-200 | {"num": 2, "type": "IAE V2500"}
(1 строка)
```

Чтобы получить еще более детальные сведения, например, о типе двигателей, нужно учитывать, что созданный JSON-объект имеет сложную структуру: он содержит вложенный JSON-объект. Поэтому нужно использовать оператор «#>» для указания пути доступа к ключу второго уровня.

```
SELECT model, specifications #> '{ engines, type }'
  FROM aircrafts
  WHERE aircraft_code = '320';
```

```
    model        |  ?column?
-----------------+-------------
 Airbus A320-200 | "IAE V2500"
(1 строка)
```

**Задание.** Подумайте, какие еще таблицы было бы целесообразно дополнить столбцами типа json/jsonb. Вспомните, что, например, в таблице «Билеты» (tickets) уже есть столбец такого типа — contact_data. Выполните модификации таблиц и измените в них одну-две строки для проверки правильности ваших решений.

# 6 Запросы

Эта глава будет самой насыщенной и интересной, поскольку умение писать SQL-запросы — это не только ремесло, но, пожалуй, и искусство тоже.

В предыдущих главах мы уже не раз использовали команду SELECT и формировали с ее помощью различные запросы. Эти запросы строились как на основе одной таблицы, так и на основе двух и более таблиц. Мы рассмотрели простые способы сортировки и группировки строк в полученных выборках из таблиц, использовали функцию count для подсчета числа выбранных строк. Таким образом, вы уже получили элементарное представление о том, как формировать выборки из базы данных. В этой главе мы покажем более сложные способы их получения.

С целью приведения в систему тех знаний о формировании выборок, что были получены в предыдущих главах, в этой главе мы повторим некоторые сведения, но сделаем это уже на новых примерах.

## 6.1 Дополнительные возможности команды SELECT

Основой для экспериментов в этом разделе будут самые маленькие (по числу строк) таблицы базы данных «Авиаперевозки»: «Самолеты» (aircrafts) и «Аэропорты» (airports).

Прежде чем перейти к конкретным запросам, просто просмотрите содержимое этих двух таблиц. Таблица «Самолеты» совсем маленькая, а таблица «Аэропорты» содержит чуть больше ста строк. Для ее просмотра можно включить расширенный режим вывода данных \x.

```
SELECT * FROM aircrafts;
SELECT * FROM airports;
```

Начнем с различных условий отбора строк в предложении WHERE. Эти условия могут конструироваться с использованием следующих **операторов сравнения:** =, <>, >, >=, <, <=. В предыдущих главах мы уже использовали ряд таких операторов, поэтому сейчас рассмотрим некоторые другие способы осуществления отбора строк.

Для начала поставим перед собой такую задачу: выбрать все самолеты компании Airbus. В этом нам поможет оператор поиска **шаблонов LIKE**:

```
SELECT * FROM aircrafts WHERE model LIKE 'Airbus%';
```

Обратите внимание на символ «%», имеющий специальное значение. Он соответствует любой последовательности символов, т. е. вместо него могут быть подставлены любые символы в любом количестве, а может и не быть подставлено ни одного символа. В результате будут выбраны строки, в которых значения атрибута model начинаются с символов «Airbus»:

```
 aircraft_code |      model      | range
---------------+-----------------+-------
 320           | Airbus A320-200 | 5700
 321           | Airbus A321-200 | 5600
 319           | Airbus A319-100 | 6700
(3 строки)
```

Шаблон в операторе LIKE всегда покрывает *всю* анализируемую строку. Поэтому если требуется отыскать некоторую последовательность символов где-то внутри строки, то шаблон должен начинаться и завершаться символом «%». Однако в этом случае нужно учитывать следующие соображения. Если по тому столбцу, к которому применяется оператор LIKE, создан индекс для ускорения доступа к данным, то при наличии символа «%» в начале шаблона этот индекс использоваться не будет. В результате может ухудшиться производительность, т. е. запрос будет выполняться медленнее. Индексы подробно рассматриваются в главе 8, а вопросы производительности — в главе 10.

Конечно, существует и оператор NOT LIKE. Например, если мы захотим узнать, какими самолетами, кроме машин компаний Airbus и Boeing, располагает наша авиакомпания, то придется усложнить условие:

```
SELECT * FROM aircrafts
  WHERE model NOT LIKE 'Airbus%'
    AND model NOT LIKE 'Boeing%';
```

```
 aircraft_code |       model         | range
---------------+---------------------+-------
 SU9           | Sukhoi SuperJet-100 | 3000
 CN1           | Cessna 208 Caravan  | 1200
 CR2           | Bombardier CRJ-200  | 2700
(3 строки)
```

Кроме символа «%» в шаблоне может использоваться и символ «_», который соответствует в точности одному любому символу. В качестве примера найдем в таблице «Аэропорты» (airports) те из них, которые имеют названия длиной три символа (буквы). С этой целью зададим в качестве шаблона строку, состоящую из трех символов «_».

```
SELECT * FROM airports WHERE airport_name LIKE '___';
```

```
--[ RECORD 1 ]+------------------
 airport_code | UFA
 airport_name | Уфа
 city         | Уфа
 longitude    | 55.874417
 latitude     | 54.557511
 timezone     | Asia/Yekaterinburg
```

Существует ряд операторов для работы с **регулярными выражениями** POSIX. Эти операторы имеют больше возможностей, чем оператор LIKE. Для того чтобы выбрать, например, самолеты компаний Airbus и Boeing, можно сделать так:

```
SELECT * FROM aircrafts WHERE model ~ '^(A|Boe)';
```

```
 aircraft_code |      model      | range
---------------+-----------------+-------
 773           | Boeing 777-300  | 11100
 763           | Boeing 767-300  |  7900
 320           | Airbus A320-200 |  5700
 321           | Airbus A321-200 |  5600
 319           | Airbus A319-100 |  6700
 733           | Boeing 737-300  |  4200
(6 строк)
```

Оператор ~ ищет совпадение с шаблоном с учетом регистра символов. Символ «^» в начале регулярного выражения означает, что поиск совпадения будет привязан к началу строки. Если же требуется проверить наличие такого символа *в составе* строки, то перед ним нужно поставить символ обратной косой черты «\». Выражение в круглых скобках означает альтернативный выбор между значениями, разделяемыми символом «|». Поэтому в выборку попадут значения, начинающиеся либо на «A», либо на «Boe».

Для инвертирования смысла оператора ~ нужно перед ним добавить знак «!». В качестве примера отыщем модели самолетов, которые не завершаются числом 300.

```
SELECT * FROM aircrafts WHERE model !~ '300$';
```

В этом регулярном выражении символ «$» означает привязку поискового шаблона к концу строки. Если же требуется проверить наличие такого символа *в составе* строки, то перед ним нужно поставить символ обратной косой черты «\».

```
 aircraft_code |       model         | range
---------------+---------------------+-------
 SU9           | Sukhoi SuperJet-100 | 3000
 320           | Airbus A320-200     | 5700
 321           | Airbus A321-200     | 5600
 319           | Airbus A319-100     | 6700
 CN1           | Cessna 208 Caravan  | 1200
 CR2           | Bombardier CRJ-200  | 2700
(6 строк)
```

Использование регулярных выражений подробно рассматривается в разделе документации 9.7.3 «Регулярные выражения POSIX».

В качестве замены традиционных операторов сравнения могут использоваться **предикаты сравнения**, которые ведут себя так же, как и операторы, но имеют другой синтаксис.

Давайте ответим на вопрос: какие самолеты имеют дальность полета в диапазоне от 3000 км до 6000 км? Ответ получим с помощью предиката BETWEEN.

```
SELECT * FROM aircrafts WHERE range BETWEEN 3000 AND 6000;
```

```
 aircraft_code |        model        | range
---------------+---------------------+-------
 SU9           | Sukhoi SuperJet-100 | 3000
 320           | Airbus A320-200     | 5700
 321           | Airbus A321-200     | 5600
 733           | Boeing 737-300      | 4200
(4 строки)
```

Обратите внимание, что граничное значение 3000 было включено в полученную выборку.

При выборке данных можно проводить вычисления и получать в результирующей таблице **вычисляемые столбцы**. Если мы захотим представить дальность полета лайнеров не только в километрах, но и в милях, то нужно вычислить это выражение и для удобства присвоить новому столбцу псевдоним с помощью ключевого слова AS.

```
SELECT model, range, range / 1.609 AS miles FROM aircrafts;
```

```
     model           | range |        miles
---------------------+-------+----------------------
 Boeing 777-300      | 11100 | 6898.6948415164698571
 Boeing 767-300      |  7900 | 4909.8819142324425109
...
(9 строк)
```

По всей вероятности, такая высокая точность представления значений в милях не требуется, поэтому мы можем уменьшить ее до разумного предела в два десятичных знака:

```
SELECT model, range, round( range / 1.609, 2 ) AS miles
   FROM aircrafts;
```

```
       model         | range |  miles
---------------------+-------+---------
 Boeing 777-300      | 11100 | 6898.69
 Boeing 767-300      |  7900 | 4909.88
...
```

Теперь обратимся к такому вопросу, как **упорядочение строк** при выводе. Если не принять специальных мер, то СУБД не гарантирует никакого конкретного порядка строк в результирующей выборке. Для упорядочения строк служит **предложение ORDER BY**, которое мы уже использовали ранее. Однако мы не говорили, что можно задать не только возрастающий, но также и убывающий порядок сортировки. Например, если мы захотим разместить самолеты в порядке убывания дальности их полета, то нужно сделать так:

```
SELECT * FROM aircrafts ORDER BY range DESC;
```

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 773           | Boeing 777-300       | 11100
 763           | Boeing 767-300       |  7900
...
 CR2           | Bombardier CRJ-200   |  2700
 CN1           | Cessna 208 Caravan   |  1200
(9 строк)
```

Мы детально разобрались с таблицей «Самолеты» (aircrafts) и теперь обратим наше внимание на таблицу «Аэропорты» (airports). В ней есть столбец «Часовой пояс» (timezone). Давайте посмотрим, в каких различных часовых поясах располагаются аэропорты. Если сделать традиционную выборку

```
SELECT timezone FROM airports;
```

то мы получим список значений, среди которых будет много повторяющихся. Конечно, это неудобно. Для того чтобы оставить в выборке только **неповторяющиеся значения,** служит **ключевое слово DISTINCT**:

```
SELECT DISTINCT timezone FROM airports ORDER BY 1;
```

Обратите внимание, что столбец, по значениям которого будут упорядочены строки, указан не с помощью его имени, а с помощью его порядкового номера в предложении SELECT.

Получим такой результат:

```
       timezone
--------------------
 Asia/Anadyr
 Asia/Chita
 Asia/Irkutsk
 Asia/Kamchatka
 Asia/Krasnoyarsk
 Asia/Magadan
 Asia/Novokuznetsk
 Asia/Novosibirsk
 Asia/Omsk
 Asia/Sakhalin
 Asia/Vladivostok
 Asia/Yakutsk
 Asia/Yekaterinburg
 Europe/Kaliningrad
 Europe/Moscow
 Europe/Samara
 Europe/Volgograd
(17 строк)
```

Таким образом, аэропорты располагаются в семнадцати различных часовых поясах. Они описаны в базе данных часовых поясов, поддерживаемой международной организацией IANA (Internet Assigned Numbers Authority), и отличаются от традиционных географических и административных часовых поясов, число которых в России равно одиннадцати.

В таблице «Аэропорты» (airports) более ста записей. Если мы поставим задачу найти три самых восточных аэропорта, то для ее решения подошел бы такой алгоритм: отсортировать строки в таблице по убыванию значений столбца «Долгота» (longitude) и включить в выборку только первые три строки. Как отсортировать строки по убыванию значений какого-либо столбца, вы уже знаете, а для того чтобы ограничить число строк, включаемых в результирующую выборку, служит **предложение LIMIT**.

```
SELECT airport_name, city, longitude
  FROM airports
  ORDER BY longitude DESC
  LIMIT 3;
```

```
 airport_name |          city             | longitude
--------------+---------------------------+------------
 Анадырь      | Анадырь                   | 177.741483
 Елизово      | Петропавловск-Камчатский | 158.453669
```

```
 Магадан      | Магадан                  | 150.720439
(3 строки)
```

А как найти еще три аэропорта, которые находятся немного западнее первой тройки, т. е. занимают места с четвертого по шестое? Алгоритм будет почти таким же, как в первой задаче, но он будет дополнен еще одним шагом: нужно пропустить три первые строки, прежде чем начать вывод. Для пропуска строк служит **предложение OFFSET**.

```
SELECT airport_name, city, longitude
  FROM airports
  ORDER BY longitude DESC
  LIMIT 3 OFFSET 3;
```

```
 airport_name    |         city         | longitude
-----------------+----------------------+-----------
 Хомутово        | Южно-Сахалинск       | 142.717531
 Хурба           | Комсомольск-на-Амуре | 136.934
 Хабаровск-Новый | Хабаровск            | 135.188361
(3 строки)
```

В дополнение к вычисляемым столбцам, когда выводимые значения получают путем вычислений, при выборке данных из таблиц можно использовать **условные выражения**, позволяющие вывести то или иное значение в зависимости от условий. В таблице «Самолеты» (aircrafts) есть столбец «Максимальная дальность полета» (range). Мы можем дополнить вывод данных из этой таблицы столбцом «Класс самолета», имея в виду принадлежность каждого самолета к классу дальнемагистральных, среднемагистральных или ближнемагистральных судов. Для этого подойдет конструкция

```
CASE WHEN condition THEN result
    [WHEN ...]
    [ELSE result]
END
```

Воспользовавшись этой конструкцией в предложении SELECT и назначив новому столбцу имя с помощью ключевого слова AS, получим следующий запрос:

```
SELECT model, range,
  CASE WHEN range < 2000 THEN 'Ближнемагистральный'
       WHEN range < 5000 THEN 'Среднемагистральный'
       ELSE 'Дальнемагистральный'
  END AS type
  FROM aircrafts
  ORDER BY model;
```

```
        model         | range |        type
----------------------+-------+---------------------
 Airbus A319-100      |  6700 | Дальнемагистральный
 Airbus A320-200      |  5700 | Дальнемагистральный
 Airbus A321-200      |  5600 | Дальнемагистральный
 Boeing 737-300       |  4200 | Среднемагистральный
 Boeing 767-300       |  7900 | Дальнемагистральный
 Boeing 777-300       | 11100 | Дальнемагистральный
 Bombardier CRJ-200   |  2700 | Среднемагистральный
 Cessna 208 Caravan   |  1200 | Ближнемагистральный
```

```
 Sukhoi SuperJet-100 |  3000 | Среднемагистральный
(9 строк)
```

## 6.2 Соединения

В тех случаях, когда информации, содержащейся в одной таблице, недостаточно для получения требуемого результата, используют **соединение (join)** таблиц. Покажем способ выполнения соединения на примере следующего запроса: выбрать все места, предусмотренные компоновкой салона самолета Cessna 208 Caravan.

Сначала приведем SQL-команду для выполнения запроса, а потом объясним, как мы ее придумали.

```
SELECT a.aircraft_code, a.model, s.seat_no, s.fare_conditions
  FROM seats AS s
  JOIN aircrafts AS a
    ON s.aircraft_code = a.aircraft_code
  WHERE a.model ~ '^Cessna'
  ORDER BY s.seat_no;
```

В предложении WHERE мы применили регулярное выражение, хотя в данном случае можно было с таким же успехом воспользоваться и оператором LIKE или функцией substr.

```
 aircraft_code |       model        | seat_no | fare_conditions
---------------+--------------------+---------+-----------------
 CN1           | Cessna 208 Caravan | 1A      | Economy
 CN1           | Cessna 208 Caravan | 1B      | Economy
 CN1           | Cessna 208 Caravan | 2A      | Economy
 CN1           | Cessna 208 Caravan | 2B      | Economy
 CN1           | Cessna 208 Caravan | 3A      | Economy
 CN1           | Cessna 208 Caravan | 3B      | Economy
 CN1           | Cessna 208 Caravan | 4A      | Economy
 CN1           | Cessna 208 Caravan | 4B      | Economy
 CN1           | Cessna 208 Caravan | 5A      | Economy
 CN1           | Cessna 208 Caravan | 5B      | Economy
 CN1           | Cessna 208 Caravan | 6A      | Economy
 CN1           | Cessna 208 Caravan | 6B      | Economy
(12 строк)
```

Данная команда иллюстрирует **соединение двух таблиц на основе равенства значений атрибутов**.

В этой команде в предложении FROM указаны две таблицы — aircrafts и seats, причем каждая из них получила еще и псевдоним с помощью ключевого слова AS (заметим, что оно не является обязательным). Конечно, псевдонимы могут состоять не только из одной буквы, как в нашем примере. Псевдонимы удобны в тех случаях, когда в соединяемых таблицах есть одноименные атрибуты. В таких случаях в списке атрибутов, следующих за ключевым словом SELECT, необходимо указывать либо имя таблицы, из которой выбирается значение этого атрибута, либо ее псевдоним, но псевдоним может быть коротким, что удобнее при написании команды. Псевдоним

и атрибут соединяются символом «.». Псевдонимы используются и в предложениях WHERE, GROUP BY, ORDER BY, HAVING, т. е. во всех частях команды SELECT.

Итак, как мы рассуждали? Если бы в качестве исходных сведений мы получили сразу код самолета — «CN1», то запрос свелся бы к выборке из одной таблицы «Места» (seats). Он был бы таким:

```
SELECT * FROM seats WHERE aircraft_code = 'CN1';
```

Но нам дано название модели, а не ее код, поэтому придется подключить к работе и таблицу «Самолеты» (aircrafts), в которой хранятся наименования моделей. Для того чтобы решить, удовлетворяет ли строка таблицы seats поставленному условию, нужно узнать, какой модели самолета соответствует эта строка. Как это можно узнать? В каждой строке таблицы seats есть атрибут aircraft_code, такой же атрибут есть и в каждой строке таблицы aircrafts. Если с каждой строкой таблицы seats соединить такую строку таблицы aircrafts, в которой значение атрибута aircraft_code такое же, как и в строке таблицы seats, то сформированная комбинированная строка, составленная из атрибутов обеих таблиц, будет содержать не только номер места, класс обслуживания и код модели, но — что важно — и наименование модели. Поэтому с помощью условия WHERE можно будет отобрать только те результирующие строки, в которых значение атрибута model будет «Cessna 208 Caravan». А какие столбцы оставлять в списке столбцов предложения SELECT, решать нам. Даже если мы соединяем две таблицы (или более), то совершенно не обязательно в результирующий список столбцов включать столбцы всех таблиц, перечисленных в предложении FROM. Мы могли бы оставить только атрибуты таблицы seats:

```
SELECT s.seat_no, s.fare_conditions
  FROM seats s
  JOIN aircrafts a ON s.aircraft_code = a.aircraft_code
  WHERE a.model ~ '^Cessna'
  ORDER BY s.seat_no;
```

```
 seat_no | fare_conditions
---------+-----------------
 1A      | Economy
 1B      | Economy
...
(12 строк)
```

Если подвести итог, то можно упрощенно объяснить механизм построения соединения следующим образом. Сначала формируются все попарные комбинации строк из обеих таблиц, т. е. декартово произведение множеств строк этих таблиц. Эти комбинированные строки включают в себя все атрибуты обеих таблиц. Затем в дело вступает условие s.aircraft_code = a.aircraft_code. Это означает, что в результирующем множестве строк останутся только те из них, в которых значения атрибута aircraft_code, взятые из таблицы aircrafts и из таблицы seats, одинаковые. Строки, не удовлетворяющие этому критерию, отфильтровываются. Это означает на практике, что каждой строке из таблицы «Места» мы сопоставили только одну конкретную строку из таблицы «Самолеты», из которой мы теперь можем взять значение атрибута «Модель самолета», чтобы включить ее в итоговый вывод данных.

На практике описанный механизм не реализуется буквально. Специальная подсистема PostgreSQL, называемая планировщиком, строит план выполнения запроса, который является гораздо более эффективным, чем упрощенный план, представленный здесь. Детально вопросы планирования запросов рассматриваются в главе 10.

Запрос, который мы рассмотрели, можно записать немного по-другому, без использования предложения JOIN (обратите внимание, что мы не использовали ключевое слово AS для назначения псевдонимов таблицам).

```
SELECT a.aircraft_code, a.model, s.seat_no, s.fare_conditions
  FROM seats s, aircrafts a
  WHERE s.aircraft_code = a.aircraft_code
    AND a.model ~ '^Cessna'
  ORDER BY s.seat_no;
```

В этом варианте запроса условие соединения таблиц s.aircraft_code = a.aircraft_code перешло из предложения FROM в предложение WHERE, а таблицы просто перечислены в предложении FROM через запятую. Простые запросы зачастую записывают именно в такой форме, без предложения JOIN, а в предложении WHERE указывают критерии, которым должны удовлетворять результирующие строки.

Изучая язык SQL вообще и способы выполнения соединений в частности, нужно иметь в виду, что *результатом любых реляционных операций над отношениями (таблицами, представлениями) также является отношение*. Поэтому такие операции можно произвольно комбинировать друг с другом.

В соединении одна и та же таблица может участвовать дважды, т. е. формировать **соединение таблицы с самой собой**. В качестве примера рассмотрим запрос для создания представления «Рейсы» (flights_v), о котором шла речь в главе 5. Этот запрос выглядит так:

```
CREATE OR REPLACE VIEW flights_v AS
  SELECT f.flight_id,
         f.flight_no,
         f.scheduled_departure,
         timezone( dep.timezone, f.scheduled_departure )
           AS scheduled_departure_local,
         f.scheduled_arrival,
         timezone( arr.timezone, f.scheduled_arrival )
           AS scheduled_arrival_local,
         f.scheduled_arrival - f.scheduled_departure
           AS scheduled_duration,
         f.departure_airport,
         dep.airport_name AS departure_airport_name,
         dep.city AS departure_city,
         f.arrival_airport,
         arr.airport_name AS arrival_airport_name,
         arr.city AS arrival_city,
         f.status,
         f.aircraft_code,
         f.actual_departure,
         timezone( dep.timezone, f.actual_departure )
           AS actual_departure_local,
         f.actual_arrival,
```

```
       timezone( arr.timezone, f.actual_arrival )
         AS actual_arrival_local,
       f.actual_arrival - f.actual_departure AS actual_duration
FROM flights f,
     airports dep,
     airports arr
WHERE f.departure_airport = dep.airport_code
  AND f.arrival_airport = arr.airport_code;
```

В этом представлении используется не только таблица «Рейсы» (flights), но также и таблица «Аэропорты» (airports). Причем она используется, условно говоря, дважды. Поясним, что мы имеем в виду. Как вы уже знаете из главы 3, при соединении двух таблиц в результирующую выборку попадают те комбинации строк из первой и второй таблиц, которые удовлетворяют условию, указанному в предложении WHERE. Будем рассуждать от противного. Пусть в предложение FROM таблица «Аэропорты» (airports) будет указана только один раз, тогда предложения FROM и WHERE будут выглядеть так:

```
FROM flights f, airports a
WHERE f.departure_airport = a.airport_code
  AND f.arrival_airport = a.airport_code;
```

Это означает, что при соединении таблиц flights и airports PostgreSQL будет пытаться для каждой строки из таблицы flights найти такую строку в таблице airports, в которой значение атрибута airport_code будет равно не только значению атрибута departure_airport, но также и значению атрибута arrival_airport в таблице flights. Получается, что данное условие будет выполнено, если только аэропорт вылета и аэропорт назначения будет одним и тем же. Однако в сфере пассажирских авиаперевозок таких рейсов не бывает. Конечно, иногда самолеты возвращаются в пункт вылета, но это уже совсем другая ситуация, которая в нашей учебной базе данных не учитывается.

Таким образом, приходим к выводу о том, что каждую строку из таблицы «Рейсы» (flights) необходимо соединять с двумя *различными* строками из таблицы «Аэропорты»: ведь аэропорт вылета и аэропорт назначения — это *различные* аэропорты. Однако при однократном включении таблицы «Аэропорты» (airports) в предложение FROM сделать это невозможно, поэтому поступают так: к таблице airports в предложении FROM обращаются дважды, как будто это две копии одной и той же таблицы. Конечно, на самом деле никаких копий не создается, а просто в результате поиск строк в ней будет производиться дважды: один раз для атрибута departure_airport, а второй раз — для атрибута arrival_airport. Но поскольку необходимо обеспечить однозначную идентификацию, то каждой «копии» (экземпляру) таблицы airports присваивают уникальный псевдоним, в нашем случае это dep и arr, т. е. departure и arrival. Эти псевдонимы указывают, из какой «копии» (экземпляра) таблицы airports нужно брать значение атрибута airport_code для сопоставления с атрибутами departure_airport и arrival_airport.

Рассмотрев этот пример, вновь обратимся к соединениям такого типа и покажем три способа выполнения **соединения таблицы с самой собой**, отличающиеся синтаксически, но являющиеся функционально эквивалентными. Наш запрос-иллюстрация должен выяснить: сколько всего маршрутов нужно было бы сформировать, если бы требовалось соединить каждый город со всеми остальными городами? Если в городе

имеется более одного аэропорта, то договоримся рейсы из каждого из них (в каждый из них) считать отдельными маршрутами. Поэтому правильнее было бы говорить не о маршрутах из каждого города, а о маршрутах из каждого аэропорта во все другие аэропорты. Конечно, рейсов из любого города в тот же самый город быть не должно.

Первый вариант запроса использует обычное перечисление имен таблиц в предложении FROM. Поскольку имена таблиц совпадают, используются псевдонимы. В таком случае СУБД обращается к таблице дважды, как если бы это были различные таблицы.

```
SELECT count( * )
  FROM airports a1, airports a2
  WHERE a1.city <> a2.city;
```

Как мы уже говорили ранее, СУБД соединяет каждую строку первой таблицы с каждой строкой второй таблицы, т. е. формирует **декартово произведение** таблиц — все попарные комбинации строк из двух таблиц. Затем СУБД отбрасывает те комбинированные строки, которые не удовлетворяют условию, приведенному в предложении WHERE. В нашем примере условие как раз и отражает требование о том, что рейсов из одного города в тот же самый город быть не должно.

```
 count
-------
 10704
(1 строка)
```

Во втором варианте запроса мы используем **соединение таблиц на основе неравенства значений атрибутов**. Тем самым мы перенесли условие отбора результирующих строк из предложения WHERE в предложение FROM.

```
SELECT count( * )
  FROM airports a1
  JOIN airports a2 ON a1.city <> a2.city;
```

```
 count
-------
 10704
(1 строка)
```

Третий вариант предусматривает **явное использование декартова произведения таблиц**. Для этого служит предложение CROSS JOIN. Лишние строки, как и в первом варианте, отсеиваем с помощью предложения WHERE:

```
SELECT count( * )
  FROM airports a1 CROSS JOIN airports a2
  WHERE a1.city <> a2.city;
```

```
 count
-------
 10704
(1 строка)
```

С точки зрения СУБД эти три варианта эквивалентны, они отличаются лишь синтаксисом. Для них PostgreSQL выберет один и тот же план (порядок) выполнения запроса.

Теперь обратимся к так называемым **внешним соединениям**. Предположим, что мы задались вопросом: сколько маршрутов обслуживают самолеты каждого типа? Если не требовать вывода наименований моделей самолетов, тогда всю необходимую информацию можно получить из материализованного представления «Маршруты» (routes). Но мы все же будем выводить и наименования моделей, поэтому обратимся также к таблице «Самолеты» (aircrafts). Соединим эти таблицы на основе атрибута aircraft_code, сгруппируем строки и просто воспользуемся функцией count. В этом запросе внешнее соединение еще не используется.

```
SELECT r.aircraft_code, a.model, count( * ) AS num_routes
  FROM routes r
  JOIN aircrafts a ON r.aircraft_code = a.aircraft_code
  GROUP BY 1, 2
  ORDER BY 3 DESC;
```

```
 aircraft_code |        model         | num_routes
---------------+----------------------+------------
 CR2           | Bombardier CRJ-200   |        232
 CN1           | Cessna 208 Caravan   |        170
 SU9           | Sukhoi SuperJet-100  |        158
 319           | Airbus A319-100      |         46
 733           | Boeing 737-300       |         36
 321           | Airbus A321-200      |         32
 763           | Boeing 767-300       |         26
 773           | Boeing 777-300       |         10
(8 строк)
```

Обратите внимание, что в таблице «Самолеты» (aircrafts) представлено 9 моделей, а в этой выборке лишь 8 строк. Значит, какая-то модель самолета не участвует в выполнении рейсов. Как ее выявить? С помощью такого запроса:

```
SELECT a.aircraft_code AS a_code,
       a.model,
       r.aircraft_code AS r_code,
       count( r.aircraft_code ) AS num_routes
  FROM aircrafts a
  LEFT OUTER JOIN routes r ON r.aircraft_code = a.aircraft_code
  GROUP BY 1, 2, 3
  ORDER BY 4 DESC;
```

```
 a_code |        model         | r_code | num_routes
--------+----------------------+--------+------------
 CR2    | Bombardier CRJ-200   | CR2    |        232
 CN1    | Cessna 208 Caravan   | CN1    |        170
 SU9    | Sukhoi SuperJet-100  | SU9    |        158
 319    | Airbus A319-100      | 319    |         46
 733    | Boeing 737-300       | 733    |         36
 321    | Airbus A321-200      | 321    |         32
 763    | Boeing 767-300       | 763    |         26
 773    | Boeing 777-300       | 773    |         10
```

```
 320    | Airbus A320-200     |        |          0
(9 строк)
```

В данном запросе используется **левое внешнее соединение** — об этом говорит предложение LEFT OUTER JOIN. В качестве базовой таблицы выбирается таблица aircrafts, указанная в запросе слева от предложения LEFT OUTER JOIN, и для каждой строки, находящейся в ней, из таблицы routes подбираются строки, в которых значение атрибута aircraft_code такое же, как и в текущей строке таблицы aircrafts. Если в таблице routes нет ни одной соответствующей строки, то при отсутствии ключевых слов LEFT OUTER результирующая комбинированная строка просто не будет сформирована и не попадет в выборку. Но при наличии ключевых слов LEFT OUTER результирующая строка все равно будет сформирована. Это происходит таким образом: если для строки из левой таблицы (левой относительно предложения LEFT OUTER JOIN) не находится ни одной соответствующей строки в правой таблице, тогда в результирующую строку вместо значений столбцов правой таблицы будут помещены значения NULL. Получается, что для строки из таблицы aircrafts, в которой значение атрибута aircraft_code равно 320, в таблице routes нет ни одной строки с таким же значением этого атрибута. В результате при выводе выборки в столбце a_code, взятом из таблицы aircrafts, будет значение 320, а в столбце r_code, взятом из таблицы routes, будет значение NULL. Этот столбец включен в выборку лишь для повышения наглядности, в реальном запросе он не нужен.

Обратите внимание, что параметром функции count является столбец из таблицы routes, поэтому count и выдает число 0 для самолета с кодом 320. Если заменить его на одноименный столбец из таблицы aircrafts, тогда count выдаст 1, что будет противоречить цели нашей задачи — подсчитать число рейсов, выполняемых на самолетах каждого типа. Напомним, что если функция count в качестве параметра получает не символ «*», а имя столбца, тогда она подсчитывает число строк, в которых значение в этом столбце определено (не равно NULL).

Кроме левого внешнего соединения существует также и **правое внешнее соединение** — RIGHT OUTER JOIN. В этом случае в качестве базовой выбирается таблица, имя которой указано справа от предложения RIGHT OUTER JOIN, а механизм получения результирующих строк в случае, когда для строки базовой таблицы не находится пары во второй таблице, точно такой же, как и для левого внешнего соединения. Как сказано в документации, правое внешнее соединение является лишь синтаксическим приемом, поскольку всегда можно заменить его левым внешним соединением, поменяв при этом имена таблиц местами.

Важно учитывать, что порядок следования таблиц в предложениях LEFT (RIGHT) OUTER JOIN никак не влияет на порядок столбцов в предложении SELECT. В вышеприведенном запросе мы написали

```
SELECT a.aircraft_code AS a_code,
       a.model,
       r.aircraft_code AS r_code,
       count( r.aircraft_code ) AS num_routes
 ...
```

Но если бы нам это было нужно, то мы могли бы поменять столбцы местами:

```
SELECT r.aircraft_code AS r_code,
       a.model,
```

```
       a.aircraft_code AS a_code,
       count( r.aircraft_code ) AS num_routes
...
```

Комбинацией этих двух видов внешних соединений является **полное внешнее соединение** — FULL OUTER JOIN. В этом случае в выборку включаются строки из левой таблицы, для которых не нашлось соответствующих строк в правой таблице, и строки из правой таблицы, для которых не нашлось соответствующих строк в левой таблице.

В практической работе при выполнении выборок зачастую выполняются **многотабличные запросы,** включающие три таблицы и более. В качестве примера рассмотрим такую задачу: определить число пассажиров, не пришедших на регистрацию билетов и, следовательно, не вылетевших в пункт назначения. Будем учитывать только рейсы, у которых фактическое время вылета не пустое, т. е. рейсы, имеющие статус «Departed» или «Arrived».

```
SELECT count( * )
  FROM (
    ticket_flights t
    JOIN flights f ON t.flight_id = f.flight_id
  )
  LEFT OUTER JOIN boarding_passes b
    ON t.ticket_no = b.ticket_no AND t.flight_id = b.flight_id
  WHERE f.actual_departure IS NOT NULL AND b.flight_id IS NULL;
```

Оказывается, таких пассажиров нет.

```
 count
-------
     0
(1 строка)
```

При формировании запроса вспомним, что таблица «Посадочные талоны» (boarding_passes) связана с таблицей «Перелеты» (ticket_flights) по внешнему ключу, а тип связи — 1:1, т. е. каждой строке из таблицы ticket_flights соответствует *не более* одной строки в таблице boarding_passes: ведь строка в таблицу boarding_passes добавляется только тогда, когда пассажир прошел регистрацию на рейс. Однако теоретически, да и практически тоже, пассажир может на регистрацию не явиться, тогда строка в таблицу boarding_passes добавлена не будет.

Поскольку нас интересуют только рейсы с непустым временем вылета, нам придется обратиться к таблице «Рейсы» (flights) и соединить ее с таблицей ticket_flights по атрибуту flight_id. А затем для подключения таблицы boarding_passes мы используем левое внешнее соединение, т. к. в этой таблице может не оказаться строки, соответствующей строке из таблицы ticket_flights.

В предложении WHERE второе условие — b.flight_id IS NULL. Оно как раз и позволяет выявить те комбинированные строки, в которых столбцам таблицы boarding_passes были назначены значения NULL из-за того, что в ней не нашлось строки, для которой выполнялось бы условие t.ticket_no = b.ticket_no AND t.flight_id = b.flight_id. Конечно, мы могли использовать любой столбец таблицы boarding_passes, а не только b.flight_id, для проверки на NULL.

При формировании соединений подключение таблиц выполняется слева направо, т. е. берется самая первая таблица в предложении FROM и с ней соединяется вторая таблица, затем с полученным набором строк соединяется третья таблица и т. д. Если требуется изменить порядок соединения таблиц, то могут использоваться круглые скобки. В приведенном запросе мы использовали круглые скобки для наглядности, однако в данном случае они не были обязательными. Необходимо различать описанный выше логический порядок соединения таблиц, т. е. взгляд с позиции программиста, пишущего запрос, и тот фактический порядок выполнения запроса, который будет сформирован планировщиком. Они могут различаться. Подробно о планах выполнения запросов сказано в главе 10.

Теперь рассмотрим более сложный пример. Известно, что в компьютерных системах бывают сбои. Предположим, что возможна такая ситуация: при бронировании билета пассажир выбрал один класс обслуживания, например, «Business», а при регистрации на рейс ему выдали посадочный талон на то место в салоне самолета, где класс обслуживания — «Economy». Необходимо выявить все случаи несовпадения классов обслуживания.

Сведения о классе обслуживания, который пассажир выбрал при бронировании билета, содержатся в таблице «Перелеты» (ticket_flights). Однако в таблице «Посадочные талоны» (boarding_passes), которая «отвечает» за посадку на рейс, сведений о классе обслуживания, который пассажир получил при регистрации, нет. Эти сведения можно получить только из таблицы «Места» (seats). Причем, сделать это можно, зная код модели самолета, выполняющего рейс, и номер места в салоне самолета. Номер места можно взять из таблицы boarding_passes, а код модели самолета можно получить из таблицы «Рейсы» (flights), связав ее с таблицей boarding_passes. Для полноты информационной картины необходимо получить еще фамилию и имя пассажира из таблицы «Билеты» (tickets), связав ее с таблицей ticket_flights по атрибуту «Номер билета» (ticket_no). При формировании запроса выберем в качестве, условно говоря, базовой таблицы таблицу boarding_passes, а затем будем поэтапно подключать остальные таблицы. В предложении WHERE будет только одно условие: несовпадение требуемого и фактического классов обслуживания.

В результате получим запрос, включающий пять таблиц.

```
SELECT f.flight_no,
       f.scheduled_departure,
       f.flight_id,
       f.departure_airport,
       f.arrival_airport,
       f.aircraft_code,
       t.passenger_name,
       tf.fare_conditions AS fc_to_be,
       s.fare_conditions AS fc_fact,
       b.seat_no
  FROM boarding_passes b
  JOIN ticket_flights tf
    ON b.ticket_no = tf.ticket_no AND b.flight_id = tf.flight_id
  JOIN tickets t ON tf.ticket_no = t.ticket_no
  JOIN flights f ON tf.flight_id = f.flight_id
  JOIN seats s
    ON b.seat_no = s.seat_no AND f.aircraft_code = s.aircraft_code
```

```
 WHERE tf.fare_conditions <> s.fare_conditions
 ORDER BY f.flight_no, f.scheduled_departure;
```

Этот запрос не выдаст ни одной строки, значит, пассажиров, получивших при регистрации неправильный класс обслуживания, не было.

Чтобы все же удостовериться в работоспособности этого запроса, можно в таблице boarding_passes изменить в одной строке номер места таким образом, чтобы этот пассажир переместился из салона экономического класса в салон бизнес-класса.

```
UPDATE boarding_passes
  SET seat_no = '1A'
  WHERE flight_id = 1 AND seat_no = '17A';
```

```
UPDATE 1
```

Выполним запрос еще раз. Теперь он выдаст одну строку.

```
--[ RECORD 1 ]-------+----------------------
 flight_no           | PG0405
 scheduled_departure | 2016-09-13 13:35:00+08
 flight_id           | 1
 departure_airport   | DME
 arrival_airport     | LED
 aircraft_code       | 321
 passenger_name      | PAVEL AFANASEV
 fc_to_be            | Economy
 fc_fact             | Business
 seat_no             | 1A
```

В предложении FROM можно использовать виртуальные таблицы, сформированные с помощью **ключевого слова VALUES**.

Предположим, что для выработки финансовой стратегии нашей авиакомпании требуется следующая информация: распределение количества бронирований по диапазонам сумм с шагом в сто тысяч рублей. Максимальная сумма в одном бронировании составляет 1 204 500 рублей. Учтем это при формировании диапазонов стоимостей.

Виртуальной таблице, создаваемой с помощью ключевого слова VALUES, присваивают имя с помощью ключевого слова AS. После имени в круглых скобках приводится список имен столбцов этой таблицы.

```
SELECT r.min_sum, r.max_sum, count( b.* )
  FROM bookings b
  RIGHT OUTER JOIN
   ( VALUES ( 0, 100000 ),        ( 100000, 200000 ),
            ( 200000, 300000 ),   ( 300000, 400000 ),
            ( 400000, 500000 ),   ( 500000, 600000 ),
            ( 600000, 700000 ),   ( 700000, 800000 ),
            ( 800000, 900000 ),   ( 900000, 1000000 ),
            ( 1000000, 1100000 ), ( 1100000, 1200000 ),
            ( 1200000, 1300000 )
   ) AS r ( min_sum, max_sum )
   ON b.total_amount >= r.min_sum AND b.total_amount < r.max_sum
  GROUP BY r.min_sum, r.max_sum
  ORDER BY r.min_sum;
```

В этом запросе мы использовали внешнее соединение. Сделано это для того, чтобы в случаях, когда в каком-то диапазоне не окажется ни одного бронирования, результирующая строка выборки все же была бы сформирована. А правое соединение было выбрано только потому, что в качестве первой, базовой, таблицы мы выбрали таблицу «Бронирования» (bookings), но именно в ней может не оказаться ни одной строки для соединения с какой-либо строкой виртуальной таблицы. А все строки виртуальной таблицы, стоящей справа от предложения RIGHT OUTER JOIN, должны быть обязательно представлены в выборке: это позволит сразу увидеть «пустые» диапазоны, если они будут.

В этом запросе можно использовать и левое внешнее соединение, если поменять таблицы местами.

```
 min_sum | max_sum |  count
---------+---------+--------
       0 |  100000 | 198314
  100000 |  200000 |  46943
  200000 |  300000 |  11916
  300000 |  400000 |   3260
  400000 |  500000 |   1357
  500000 |  600000 |    681
  600000 |  700000 |    222
  700000 |  800000 |     55
  800000 |  900000 |     24
  900000 | 1000000 |     11
 1000000 | 1100000 |      4
 1100000 | 1200000 |      0
 1200000 | 1300000 |      1
(13 строк)
```

Обратите внимание, что для диапазона от 1100 до 1200 тысяч рублей значение счетчика бронирований равно нулю. Если бы мы не использовали внешнее соединение, то эта строка вообще не попала бы в выборку. Конечно, информация была бы получена та же самая, но воспринимать ее было бы сложнее.

В команде SELECT предусмотрены средства для выполнения операций с выборками, как с множествами, а именно:

- предложение UNION предназначено для вычисления объединения множеств строк из двух выборок;
- предложение INTERSECT предназначено для вычисления пересечения множеств строк из двух выборок;
- предложение EXCEPT предназначено для вычисления разности множеств строк из двух выборок.

Запросы должны возвращать одинаковое число столбцов, типы данных у столбцов также должны совпадать.

Рассмотрим эти операции, используя материализованное представление «Маршруты» (routes).

Начнем с операции **объединения множеств строк — UNION**. Строка включается в итоговое множество (выборку), если она присутствует хотя бы в одном из них.

Строки-дубликаты в результирующее множество не включаются. Для их включения нужно использовать UNION ALL.

Вопрос: в какие города можно улететь либо из Москвы, либо из Санкт-Петербурга?

```
SELECT arrival_city FROM routes
  WHERE departure_city = 'Москва'
UNION
SELECT arrival_city FROM routes
  WHERE departure_city = 'Санкт-Петербург'
ORDER BY arrival_city;
```

```
     arrival_city
-------------------------
 Абакан
 Анадырь
 Анапа
 ...
 Южно-Сахалинск
 Якутск
 Ярославль
(87 строк)
```

Рассмотрим операцию **пересечения множеств строк — INTERSECT**. Строка включается в итоговое множество (выборку), если она присутствует в каждом из них. Строки-дубликаты в результирующее множество не включаются. Для их включения нужно использовать INTERSECT ALL.

Вопрос: в какие города можно улететь как из Москвы, так и из Санкт-Петербурга?

```
SELECT arrival_city FROM routes
  WHERE departure_city = 'Москва'
INTERSECT
SELECT arrival_city FROM routes
  WHERE departure_city = 'Санкт-Петербург'
ORDER BY arrival_city;
```

```
  arrival_city
-----------------
 Воркута
 Воронеж
 Казань
 ...
 Чебоксары
 Элиста
(15 строк)
```

В завершение рассмотрим операцию получения **разности множеств строк — EXCEPT**. Строка включается в итоговое множество (выборку), если она присутствует в первом множестве (выборке), но отсутствует во втором. Строки-дубликаты в результирующее множество не включаются. Для их включения нужно использовать EXCEPT ALL.

Вопрос: в какие города можно улететь из Санкт-Петербурга, но нельзя из Москвы?

```
SELECT arrival_city FROM routes
  WHERE departure_city = 'Санкт-Петербург'
EXCEPT
SELECT arrival_city FROM routes
  WHERE departure_city = 'Москва'
ORDER BY arrival_city;
```

```
 arrival_city
--------------
 Иркутск
 Калуга
 Москва
 Удачный
 Череповец
 Якутск
 Ярославль
(7 строк)
```

Конечно, при выполнении этих операций можно соединять не только две таблицы, но и большее их число. При этом нужно либо учитывать приоритеты выполнения операций, либо использовать скобки. Согласно документации, INTERSECT связывает свои подзапросы сильнее, чем UNION, а EXCEPT связывает свои подзапросы так же сильно, как UNION.

## 6.3 Агрегирование и группировка

Среди множества функций, имеющихся в PostgreSQL, важное место занимают агрегатные функции. С одной из них, функцией count, мы уже работали довольно много. Давайте рассмотрим еще ряд функций из этой группы и сделаем это на примере таблицы «Бронирования» (bookings).

Для расчета среднего значения по столбцу используется функция avg (от слова average).

```
SELECT avg( total_amount ) FROM bookings;
```

```
       avg
--------------------
 79025.605811528685
(1 строка)
```

Для получения максимального значения по столбцу используется функция max.

```
SELECT max( total_amount ) FROM bookings;
```

```
    max
------------
 1204500.00
(1 строка)
```

Для получения минимального значения по столбцу используется функция min.

```
SELECT min( total_amount ) FROM bookings;
```

```
   min
---------
 3400.00
(1 строка)
```

Мы уже много раз выполняли **группировку строк** в выборке при помощи предложения GROUP BY, поэтому рассмотрим только два примера.

Первый будет таким: давайте подсчитаем, сколько маршрутов предусмотрено из Москвы в другие города. При формировании запроса не будем учитывать частоту рейсов в неделю, т. е. независимо от того, выполняется какой-то рейс один раз в неделю или семь раз, он учитывается только однократно. Воспользуемся материализованным представлением «Маршруты» (routes).

```
SELECT arrival_city, count( * )
  FROM routes
  WHERE departure_city = 'Москва'
  GROUP BY arrival_city
  ORDER BY count DESC;
```

```
 arrival_city             | count
--------------------------+-------
 Санкт-Петербург          |    12
 Брянск                   |     9
 Ульяновск                |     5
 Йошкар-Ола               |     4
 Петрозаводск             |     4
 ...
```

В качестве второго примера рассмотрим ситуацию, когда руководству компании потребовалась обобщенная информация по частоте выполнения рейсов, а именно: сколько рейсов выполняется ежедневно, сколько рейсов — шесть дней в неделю, пять и т. д. Опять обратимся к материализованному представлению «Маршруты» (routes). Но теперь при формировании запроса, в отличие от первого примера, воспользуемся столбцом days_of_week, в котором содержатся *массивы* номеров дней недели, когда выполняется данный рейс.

```
SELECT array_length( days_of_week, 1 ) AS days_per_week,
       count( * ) AS num_routes
  FROM routes
  GROUP BY days_per_week
  ORDER BY 1 desc;
```

```
 days_per_week | num_routes
---------------+------------
             7 |        482
             3 |         54
             2 |         88
             1 |         86
(4 строки)
```

В этом запросе используется функция array_length, возвращающая количество элементов в указанном измерении массива. Поскольку массив одномерный, то вторым параметром функции будет число 1 — первое измерение.

При выполнении выборок можно с помощью условий, заданных в предложении WHERE, сузить множество выбираемых строк. Аналогичная возможность существует и при выполнении группировок: можно включить в результирующее множество не все строки, а лишь те, которые удовлетворяют некоторому условию. Это условие можно задать в **предложении HAVING**. Важно помнить, что предложение WHERE работает с отдельными строками еще до выполнения группировки с помощью GROUP BY, а предложение HAVING — уже после выполнения группировки.

В качестве примера приведем такой запрос: определить, сколько существует маршрутов из каждого города в другие города, и вывести названия городов, из которых в другие города существует не менее 15 маршрутов.

```
SELECT departure_city, count( * )
  FROM routes
  GROUP BY departure_city
  HAVING count( * ) >= 15
  ORDER BY count DESC;
```

```
 departure_city  | count
-----------------+-------
 Москва          |   154
 Санкт-Петербург |    35
 Новосибирск     |    19
 Екатеринбург    |    15
(4 строки)
```

В подавляющем большинстве городов только один аэропорт, но есть и такие города, в которых более одного аэропорта. Давайте их выявим.

```
SELECT city, count( * )
  FROM airports
  GROUP BY city
  HAVING count( * ) > 1;
```

```
 city      | count
-----------+-------
 Ульяновск |     2
 Москва    |     3
(2 строки)
```

Кроме обычных агрегатных функций существуют и так называемые **оконные функции (window functions)**, технология использования которых описана в документации в разделе 3.5 «Оконные функции». Эти функции предоставляют возможность производить вычисления на множестве строк, логически связанных с текущей строкой, т. е. имеющих то или иное отношение к ней.

При работе с оконными функциями используются концепции *раздела* (partition) и *оконного кадра* (window frame). Сначала объясним эти понятия на примере. Предположим, что руководство нашей компании хочет усовершенствовать тарифную политику и с этой целью просит нас предоставить сведения о распределении количества проданных билетов на некоторые рейсы во времени. Количество проданных билетов должно выводиться в виде накопленного показателя, суммирование должно производиться в пределах каждого календарного месяца. Форма отчета предположительно должна быть такой:

```
 book_ref |        book_date       | month | day | count
----------+------------------------+-------+-----+-------
 A60039   | 2016-08-22 12:02:00+08 |     8 |  22 |     1
 554340   | 2016-08-23 23:04:00+08 |     8 |  23 |     2
 854C4C   | 2016-08-24 10:52:00+08 |     8 |  24 |     5
 854C4C   | 2016-08-24 10:52:00+08 |     8 |  24 |     5
 854C4C   | 2016-08-24 10:52:00+08 |     8 |  24 |     5
 81D8AF   | 2016-08-25 10:22:00+08 |     8 |  25 |     6
 ...
 8D6873   | 2016-08-31 17:09:00+08 |     8 |  31 |    59
 E82829   | 2016-08-31 20:56:00+08 |     8 |  31 |    60
 ECA0D7   | 2016-09-01 00:48:00+08 |     9 |   1 |     1
 E3BD32   | 2016-09-01 04:44:00+08 |     9 |   1 |     2
 ...
 EB11BB   | 2016-09-03 12:02:00+08 |     9 |   3 |    14
 19FE38   | 2016-09-03 17:42:00+08 |     9 |   3 |    16
 19FE38   | 2016-09-03 17:42:00+08 |     9 |   3 |    16
 536A3D   | 2016-09-03 19:19:00+08 |     9 |   3 |    18
 536A3D   | 2016-09-03 19:19:00+08 |     9 |   3 |    18
 02E6B6   | 2016-09-04 01:39:00+08 |     9 |   4 |    19
(79 строк)
```

Для примера был выбран рейс с идентификатором 1.

В столбцах book_ref и book_date приводятся номер бронирования и момент времени, когда оно было произведено. В столбцах month и day указываются порядковый номер месяца и день этого месяца. В столбце count содержатся суммарные (накопленные) количества билетов, проданных на каждый момент времени. С первого дня нового месяца подсчет числа проданных билетов начинается сначала. Таким образом, в нашем примере в качестве раздела (partition) будет выступать множество строк, у которых даты продажи билета (т. е. даты бронирования) относятся к одному и тому же месяцу. В результате в полученной выборке будет сформировано два раздела.

Понятие оконного кадра (window frame) является важным, поскольку многие оконные функции работают не со всеми строками раздела, а только с теми, которые образуют оконный кадр текущей строки. Если строки в разделе не упорядочены, то оконным кадром текущей строки по умолчанию считается множество всех строк раздела. Однако в том случае, когда строки в разделе упорядочены по какому-то критерию, тогда в состав оконного кадра по умолчанию включаются строки, начиная с первой строки раздела и заканчивая текущей строкой. Если же существуют строки, имеющие такое же значение критерия сортировки, что и текущая строка, и расположенные *после* нее, то они также включаются в состав оконного кадра текущей строки.

Обратите внимание на первые строки в представленной выборке. В строках с третьей по пятую значения в столбце count одинаковые и равны 5. Равенство значений имеет следующее объяснение. В рамках одного бронирования с номером «854C4C» были проданы сразу три билета на этот рейс, поэтому в этих трех строках значения в столбце book_date одинаковые. Строки в выборке упорядочены по значениям столбца book_date. Таким образом, для каждой из этих трех строк, т. е. для третьей, четвертой и пятой, значения критерия сортировки одинаковые, поэтому оконным кадром для каждой из них будут являться первые пять строк первого раздела выборки. Подсчет числа проданных билетов выполняется в пределах оконного кадра. В результате и появляется значение 5 в каждой из этих трех строк, а значений 3 и 4 нет вообще.

В приведенной выборке отражены также и случаи одновременного бронирования двух билетов на данный рейс. Вы можете найти соответствующие строки самостоятельно.

Теперь посмотрим, с помощью какого запроса был получен этот результат, и на его примере объясним синтаксические конструкции, используемые для работы с оконными функциями.

```
SELECT b.book_ref,
       b.book_date,
       extract( 'month' from b.book_date ) AS month,
       extract( 'day'   from b.book_date ) AS day,
       count( * ) OVER (
         PARTITION BY date_trunc( 'month', b.book_date )
         ORDER BY b.book_date
       ) AS count
  FROM ticket_flights tf
  JOIN tickets  t ON tf.ticket_no = t.ticket_no
  JOIN bookings b ON t.book_ref = b.book_ref
  WHERE tf.flight_id = 1
  ORDER BY b.book_date;
```

Рассмотрим конструкцию, предназначенную для вызова оконной функции:

```
  count( * ) OVER (
    PARTITION BY date_trunc( 'month', b.book_date )
    ORDER BY b.book_date
  ) AS count
```

В этой конструкции обязательным является ключевое слово OVER. Функция count — это обычная агрегатная функция, но если вслед за ней идет это ключевое слово, то она становится оконной функцией. Предложение PARTITION BY задает правило разбиения строк выборки на разделы. Предложение ORDER BY предписывает порядок сортировки строк в разделах.

Обобщая приведенные объяснения, можно сказать, что раздел включает в себя все строки выборки, имеющие в некотором смысле одинаковые свойства, например, одинаковые значения определенных выражений, задаваемых с помощью предложения PARTITION BY. Это могут быть выражения, построенные на основе одного или нескольких столбцов таблицы (или таблиц, участвующих в соединении). Оконный кадр состоит из подмножества строк данного раздела и привязан к текущей строке. Для определения границ кадра важным является наличие предложения ORDER BY при формировании раздела. В рассмотренном примере границы оконного кадра определялись по умолчанию. Однако для указания этих границ предусмотрены различные способы. Подробно о них сказано в разделе документации 4.2.8 «Вызовы оконных функций».

Не только функция count, но и другие агрегатные функции (например, sum, avg) тоже могут применяться в качестве оконных функций. Полный перечень собственно оконных функций приведен в документации в разделе 9.21 «Оконные функции».

Оконные функции, в отличие от обычных агрегатных функций, не требуют группировки строк, а работают на уровне отдельных (несгруппированных) строк. Однако

если в запросе присутствуют предложения GROUP BY и HAVING, тогда оконные функции вызываются уже *после* них. В таком случае оконные функции будут работать со строками, являющимися результатом группировки.

Рассмотрим еще один пример. Покажем, как с помощью оконной функции rank можно проранжировать аэропорты в пределах каждого часового пояса на основе их географической широты. Причем будем присваивать более высокий ранг тому аэропорту, который находится севернее.

```
SELECT airport_name, city,
       round( latitude::numeric, 2 ) AS ltd, timezone,
       rank() OVER (
          PARTITION BY timezone
          ORDER BY latitude DESC
       )
  FROM airports
  WHERE timezone IN ( 'Asia/Irkutsk', 'Asia/Krasnoyarsk' )
  ORDER BY timezone, rank;
```

В этом запросе в предложении OVER ( PARTITION BY timezone ... ) указывается, что строки относятся к одному разделу на основе совпадения значений в столбце timezone. Обратите внимание, что хотя в предложении OVER задан порядок сортировки, действующий в пределах каждого окна, тем не менее, с помощью предложения ORDER BY указан также и порядок сортировки на уровне всего запроса.

```
 airport_name  |      city     |  ltd  |    timezone      | rank
---------------+---------------+-------+------------------+------
 Усть-Илимск   | Усть-Илимск   | 58.14 | Asia/Irkutsk     |    1
 Усть-Кут      | Усть-Кут      | 56.85 | Asia/Irkutsk     |    2
 Братск        | Братск        | 56.37 | Asia/Irkutsk     |    3
 Иркутск       | Иркутск       | 52.27 | Asia/Irkutsk     |    4
 Байкал        | Улан-Удэ      | 51.81 | Asia/Irkutsk     |    5
 Норильск      | Норильск      | 69.31 | Asia/Krasnoyarsk |    1
 Стрежевой     | Стрежевой     | 60.72 | Asia/Krasnoyarsk |    2
 Богашёво      | Томск         | 56.38 | Asia/Krasnoyarsk |    3
 Емельяново    | Красноярск    | 56.18 | Asia/Krasnoyarsk |    4
 Абакан        | Абакан        | 53.74 | Asia/Krasnoyarsk |    5
 Барнаул       | Барнаул       | 53.36 | Asia/Krasnoyarsk |    6
 Горно-Алтайск | Горно-Алтайск | 51.97 | Asia/Krasnoyarsk |    7
 Кызыл         | Кызыл         | 51.67 | Asia/Krasnoyarsk |    8
(13 строк)
```

Усложним запрос — для каждого аэропорта будем вычислять разницу между его географической широтой и широтой, на которой находится самый северный аэропорт в этом же часовом поясе. Поскольку в запросе используются три конструкции с оконными функциями и при этом способ формирования разделов и порядок сортировки строк в разделах один и тот же, то вводится предложение WINDOW. Оно позволяет создать определение раздела, а затем ссылаться на него при вызове оконных функций. Самый северный аэропорт в каждом часовом поясе, т. е. самая первая строка в каждом разделе, выбирается с помощью оконной функции first_value. Строго говоря, эта функция получает доступ к первой строке оконного кадра, а не раздела. Однако когда используются правила формирования оконного кадра по умолчанию, тогда его начало совпадает с началом раздела.

Обратите внимание, что в этом запросе в каждой конструкции OVER используется ссылка на одно и то же окно, т. е. имеет место один и тот же порядок разбиения на разделы и сортировки строк, поэтому данные будут обработаны за один проход по таблице.

```
SELECT
   airport_name,
   city,
   timezone,
   latitude,
   first_value( latitude )            OVER tz AS first_in_timezone,
   latitude - first_value( latitude ) OVER tz AS delta,
   rank()                             OVER tz
  FROM airports
  WHERE timezone IN ( 'Asia/Irkutsk', 'Asia/Krasnoyarsk' )
  WINDOW tz AS ( PARTITION BY timezone ORDER BY latitude DESC )
  ORDER BY timezone, rank;

...
--[ RECORD 4 ]-----+-----------------
 airport_name      | Иркутск
 city              | Иркутск
 timezone          | Asia/Irkutsk
 latitude          | 52.268028
 first_in_timezone | 58.135
 delta             | -5.866972
 rank              | 4
--[ RECORD 5 ]-----+-----------------
 airport_name      | Байкал
 city              | Улан-Удэ
 timezone          | Asia/Irkutsk
 latitude          | 51.807764
 first_in_timezone | 58.135
 delta             | -6.327236
 rank              | 5
--[ RECORD 6 ]-----+-----------------
 airport_name      | Норильск
 city              | Норильск
 timezone          | Asia/Krasnoyarsk
 latitude          | 69.311053
 first_in_timezone | 69.311053
 delta             | 0
 rank              | 1
--[ RECORD 7 ]-----+-----------------
 airport_name      | Стрежевой
 city              | Стрежевой
 timezone          | Asia/Krasnoyarsk
 latitude          | 60.716667
 first_in_timezone | 69.311053
 delta             | -8.594386
 rank              | 2
--[ RECORD 8 ]-----+-----------------
 airport_name      | Богашёво
 city              | Томск
```

```
 timezone           | Asia/Krasnoyarsk
 latitude           | 56.380278
 first_in_timezone  | 69.311053
 delta              | -12.930775
 rank               | 3
...
```

Более подробно использование оконных функций описано в документации. Мы рекомендуем начать с раздела 3.5 «Оконные функции», в котором приводятся примеры их использования. В разделе 9.21 «Оконные функции» приводятся описания всех оконных функций, предлагаемых PostgreSQL. В разделе 4.2.8 «Вызовы оконных функций» детально рассматривается синтаксис вызова оконных функций. В разделе 7.2.5 «Обработка оконных функций» говорится о том, на каком этапе выполнения запроса производится обработка этих функций.

## 6.4 Подзапросы

Прежде чем приступить к рассмотрению столь сложной темы, как подзапросы, опишем, как в общем случае работает команда SELECT. Согласно описанию этой команды, приведенному в документации на PostgreSQL, дело, в несколько упрощенном виде, обстоит так.

1. Сначала вычисляются все элементы, приведенные в списке после ключевого слова FROM. Под такими элементами подразумеваются не только реальные таблицы, но также и виртуальные таблицы, создаваемые с помощью ключевого слова VALUES. Если таблиц больше одной, то формируется декартово произведение из множеств их строк. Например, в случае двух таблиц будут сформированы попарные комбинации каждой строки из одной таблицы с каждой строкой из другой таблицы. При этом в комбинированных строках сохраняются все атрибуты из каждой исходной таблицы.
2. Если в команде присутствует условие WHERE, то из полученного декартова произведения исключаются строки, которые этому условию не соответствуют. Таким образом, первоначальное множество строк, сформированное без всяких условий, сужается.
3. Если присутствует предложение GROUP BY, то результирующие строки группируются на основе совпадения значений одного или нескольких атрибутов, а затем вычисляются значения агрегатных функций. Если присутствует предложение HAVING, то оно отфильтровывает результирующие строки (группы), не удовлетворяющие критерию.
4. Ключевое слово SELECT присутствует всегда. Но в списке выражений, идущих после него, могут быть не только простые имена атрибутов, но и их комбинации, созданные с использованием арифметических и других операций, а также вызовы функций. Причем эти функции могут быть не только встроенные, но и созданные пользователем. В списке выражений не обязаны присутствовать *все*

атрибуты, представленные в строках используемых таблиц. Например, атрибуты, на основе которых формируются условия в предложении WHERE, могут отсутствовать в списке выражений после ключевого слова SELECT. Предложение SELECT DISTINCT удаляет дубликаты строк.

5. Если присутствует предложение ORDER BY, то результирующие строки сортируются на основе значений одного или нескольких атрибутов. По умолчанию сортировка производится по возрастанию значений.
6. Если присутствует предложение LIMIT или OFFSET, то возвращается только подмножество строк из выборки.

Приведенная схема описывает работу команды SELECT на логическом уровне, а на уровне реализации запросов в дело вступает планировщик, который и формирует план выполнения запроса.

А теперь перейдем непосредственно к теме этого параграфа — подзапросам.

Предположим, что сотрудникам аналитического отдела потребовалось провести статистическое исследование финансовых результатов работы авиакомпании. В качестве первого шага они решили подсчитать количество операций бронирования, в которых общая сумма превышает среднюю величину по всей выборке.

```
SELECT count( * ) FROM bookings
  WHERE total_amount >
     ( SELECT avg( total_amount ) FROM bookings );
```

```
 count
-------
 87224
(1 строка)
```

В приведенном запросе присутствует два предложения SELECT, но при этом только одно из них является главным в этом запросе, а другое представляет собой **подзапрос**. Он заключается в круглые скобки и является частью более общего запроса. Подзапросы могут присутствовать в предложениях SELECT, FROM, WHERE и HAVING, а также в предложении WITH, о котором мы расскажем позднее.

В приведенном примере в предложении WHERE используется так называемый **скалярный подзапрос**. Это означает, что в результате его выполнения возвращается только одно скалярное значение (один столбец и одна строка), с которым можно сравнивать другие скалярные значения.

Продолжим рассмотрение примеров использования подзапросов в предложении WHERE.

Если подзапрос выдает множество скалярных значений (или даже только одно), то можно использовать такой **подзапрос в предикате IN**. Этот предикат позволяет организовать проверку на предмет принадлежности какого-либо значения определенному множеству значений.

В качестве примера давайте выясним, какие маршруты существуют между городами часового пояса Asia/Krasnoyarsk. Подзапрос будет выдавать список городов из этого часового пояса, а в предложении WHERE главного запроса с помощью предиката IN

будет выполняться проверка на принадлежность города этому списку. При этом подзапрос выполняется *только один раз* для всего внешнего запроса, а не при обработке каждой строки из таблицы routes во внешнем запросе. Повторного выполнения подзапроса не требуется, т. к. его результат не зависит от значений, хранящихся в таблице routes. Такие подзапросы называются **некоррелированными**.

```
SELECT flight_no, departure_city, arrival_city
  FROM routes
  WHERE departure_city IN (
      SELECT city
        FROM airports
        WHERE timezone ~ 'Krasnoyarsk'
    )
    AND arrival_city IN (
      SELECT city
        FROM airports
        WHERE timezone ~ 'Krasnoyarsk'
    );
```

```
 flight_no | departure_city | arrival_city
-----------+----------------+--------------
 PG0070    | Абакан         | Томск
 PG0071    | Томск          | Абакан
 PG0313    | Абакан         | Кызыл
 PG0314    | Кызыл          | Абакан
 PG0653    | Красноярск     | Барнаул
 PG0654    | Барнаул        | Красноярск
(6 строк)
```

Можно сформировать множество значений для предиката IN с помощью скалярных подзапросов. Если мы захотим найти самый западный и самый восточный аэропорты и представить полученные сведения в наглядной форме, то запрос может быть таким:

```
SELECT airport_name, city, longitude
  FROM airports
  WHERE longitude IN (
   ( SELECT max( longitude ) FROM airports ),
   ( SELECT min( longitude ) FROM airports )
  )
  ORDER BY longitude;
```

```
 airport_name |    city     | longitude
--------------+-------------+------------
 Храброво     | Калининград |  20.592633
 Анадырь      | Анадырь     | 177.741483
(2 строки)
```

Конечно, в случае, когда необходимо, наоборот, исключить какие-либо значения из рассмотрения, можно использовать конструкцию NOT IN.

Иногда возникают ситуации, когда от подзапроса требуется лишь установить сам факт наличия или отсутствия строк в конкретной таблице, удовлетворяющих определенному условию, а непосредственные значения атрибутов в этих строках инте-

реса не представляют. В подобных случаях используют **предикат EXISTS** (или NOT EXISTS).

В качестве примера выясним, в какие города нет рейсов из Москвы.

```
SELECT DISTINCT a.city
  FROM airports a
  WHERE NOT EXISTS (
    SELECT * FROM routes r
      WHERE r.departure_city = 'Москва'
        AND r.arrival_city = a.city
    )
    AND a.city <> 'Москва'
  ORDER BY city;
```

В этом запросе мы не можем ограничиться только лишь материализованным представлением «Маршруты» (routes), поскольку в нем представлены лишь *существующие* маршруты. Полный список городов можно найти в таблице «Аэропорты» (airports). Для каждой строки (каждого города) из таблицы airports выполняется поиск строки в представлении routes, в которой значение атрибута arrival_city такое же, как в текущей строке таблицы airports. Если такой строки не найдено, значит, в этот город маршрута из Москвы нет.

Поскольку от подзапроса в предикате EXISTS требуется только установить факт наличия или отсутствия строк, соответствующих критерию отбора, то в документации рекомендуется вместо списка столбцов (или символа «*») в предложении SELECT делать так:

```
WHERE NOT EXISTS ( SELECT 1 FROM routes r ...
```

Обратите внимание на ключевое слово DISTINCT в запросе. Оно необходимо, т. к. кроме Москвы могут быть другие города, в которых есть более одного аэропорта. Один такой город уже существует — Ульяновск. Если не использовать DISTINCT, то, в принципе, возможно появление строк-дубликатов в выборке.

И еще одна важная деталь. В представленном запросе мы использовали так называемый **коррелированный (связанный) подзапрос**. В подзапросах такого типа присутствует ссылка (ссылки) на таблицу из внешнего запроса, как здесь:

```
WHERE ...
  AND r.arrival_city = a.city
```

В теории это означает, что подзапрос выполняется не один раз для всего внешнего запроса, а *для каждой строки*, обрабатываемой во внешнем запросе. Однако на практике важную роль играет умение планировщика (это специальная подсистема в СУБД) оптимизировать подобные запросы с тем, чтобы, по возможности, избегать выполнения подзапроса для каждой строки из внешнего запроса.

Получаем такой результат:

```
      city
---------------------
 Благовещенск
 Иваново
 ...
```

```
 Якутск
 Ярославль
(20 строк)
```

Рассмотрим использование подзапросов в предложениях SELECT, FROM и HAVING.

Предположим, что для выработки ценовой политики авиакомпании необходимо знать, как распределяются места разных классов в самолетах всех типов. Первый вариант решения этой задачи основан на включении **подзапросов в предложение SELECT**.

```
SELECT a.model,
  ( SELECT count( * )
      FROM seats s
      WHERE s.aircraft_code = a.aircraft_code
        AND s.fare_conditions = 'Business'
  ) AS business,
  ( SELECT count( * )
      FROM seats s
      WHERE s.aircraft_code = a.aircraft_code
        AND s.fare_conditions = 'Comfort'
  ) AS comfort,
  ( SELECT count( * )
      FROM seats s
      WHERE s.aircraft_code = a.aircraft_code
        AND s.fare_conditions = 'Economy'
  ) AS economy
  FROM aircrafts a
  ORDER BY 1;
```

Обратите внимание, что в этом запросе мы использовали коррелированные подзапросы. Все они ссылаются на столбец таблицы «Самолеты» (aircrafts), которая обрабатывается во внешнем запросе. Для каждой обрабатываемой строки таблицы aircrafts подсчитывается число строк в таблице seats, в которых атрибут aircraft_code имеет такое же значение, что и в строке таблицы aircrafts. Подзапросы отличаются друг от друга только условием fare_conditions. Поскольку все эти подзапросы не зависят друг от друга, то, хотя все они обращаются к таблице «Места» (seats), не требуется использовать для нее различные псевдонимы в этих подзапросах.

```
       model          | business | comfort | economy
----------------------+----------+---------+---------
 Airbus A319-100      |       20 |       0 |      96
 Airbus A320-200      |       20 |       0 |     120
 Airbus A321-200      |       28 |       0 |     142
 Boeing 737-300       |       12 |       0 |     118
 Boeing 767-300       |       30 |       0 |     192
 Boeing 777-300       |       30 |      48 |     324
 Bombardier CRJ-200   |        0 |       0 |      50
 Cessna 208 Caravan   |        0 |       0 |      12
 Sukhoi SuperJet-100  |       12 |       0 |      85
(9 строк)
```

А в этом варианте решения задачи используется **подзапрос в предложении FROM**.

```
SELECT s2.model,
  string_agg(
     s2.fare_conditions || ' (' || s2.num || ')', ', '
  )
  FROM (
    SELECT a.model,
        s.fare_conditions,
        count( * ) AS num
      FROM aircrafts a
      JOIN seats s ON a.aircraft_code = s.aircraft_code
      GROUP BY 1, 2
      ORDER BY 1, 2
  ) AS s2
  GROUP BY s2.model
  ORDER BY s2.model;
```

Подзапрос формирует временную таблицу в таком виде:

```
       model          | fare_conditions | num
----------------------+-----------------+-----
 Airbus A319-100      | Business        |  20
 Airbus A319-100      | Economy         |  96
 Airbus A320-200      | Business        |  20
 Airbus A320-200      | Economy         | 120
...
 Sukhoi SuperJet-100 | Business        |  12
 Sukhoi SuperJet-100 | Economy         |  85
(17 строк)
```

А в главном (внешнем) запросе используется агрегатная функция string_agg для формирования результирующего значения на основе сгруппированных строк. Эта функция отличается от агрегатных функций avg, min, max, sum и count тем, что она возвращает не числовое значение, а строку символов, составленную из значений атрибутов, указанных в качестве ее параметров. Эти значения берутся из сгруппированных строк.

```
        model         |           string_agg
----------------------+-----------------------------------------
 Airbus A319-100      | Business (20), Economy (96)
 Airbus A320-200      | Business (20), Economy (120)
 Airbus A321-200      | Business (28), Economy (142)
 Boeing 737-300       | Business (12), Economy (118)
 Boeing 767-300       | Business (30), Economy (192)
 Boeing 777-300       | Business (30), Comfort (48), Economy (324)
 Bombardier CRJ-200   | Economy (50)
 Cessna 208 Caravan   | Economy (12)
 Sukhoi SuperJet-100 | Business (12), Economy (85)
(9 строк)
```

В качестве еще одного примера использования подзапроса в предложении FROM решим такую задачу: получить перечень аэропортов в тех городах, в которых больше одного аэропорта.

```
SELECT aa.city, aa.airport_code, aa.airport_name
  FROM (
```

```
    SELECT city, count( * )
      FROM airports
      GROUP BY city
      HAVING count( * ) > 1
    ) AS a
  JOIN airports AS aa ON a.city = aa.city
  ORDER BY aa.city, aa.airport_name;
```

Благодаря использованию предложения HAVING, подзапрос выбирает города, в которых более одного аэропорта, и формирует временную таблицу в таком виде:

```
   city    | count
-----------+-------
 Ульяновск |     2
 Москва    |     3
(2 строки)
```

А в главном запросе выполняется соединение временной таблицы с таблицей «Аэропорты» (airports).

```
    city   | airport_code |    airport_name
-----------+--------------+---------------------
 Москва    | VKO          | Внуково
 Москва    | DME          | Домодедово
 Москва    | SVO          | Шереметьево
 Ульяновск | ULV          | Баратаевка
 Ульяновск | ULY          | Ульяновск-Восточный
(5 строк)
```

Для иллюстрации использования **подзапросов в предложении HAVING** решим такую задачу: определить число маршрутов, исходящих из тех аэропортов, которые расположены восточнее географической долготы 150°.

```
SELECT departure_airport, departure_city, count( * )
  FROM routes
  GROUP BY departure_airport, departure_city
  HAVING departure_airport IN (
    SELECT airport_code
      FROM airports
      WHERE longitude > 150
  )
  ORDER BY count DESC;
```

Подзапрос формирует список аэропортов, которые и будут отобраны с помощью предложения HAVING после выполнения группировки.

```
 departure_airport |      departure_city       | count
-------------------+---------------------------+-------
 DYR               | Анадырь                   |     4
 GDX               | Магадан                   |     3
 PKC               | Петропавловск-Камчатский  |     1
(3 строки)
```

В сложных запросах могут использоваться **вложенные подзапросы**. Это означает, что один подзапрос находится внутри другого. Давайте в качестве примера рассмотрим такую ситуацию: руководство авиакомпании хочет выяснить степень заполнения самолетов на всех рейсах, ведь отправлять полупустые самолеты не очень выгодно. Таким образом, запрос должен не только выдавать число билетов, проданных на данный рейс, и общее число мест в самолете, но должен также вычислять отношение этих двух показателей.

Вот какой запрос получился:

```
SELECT ts.flight_id,
       ts.flight_no,
       ts.scheduled_departure_local,
       ts.departure_city,
       ts.arrival_city,
       a.model,
       ts.fact_passengers,
       ts.total_seats,
       round( ts.fact_passengers::numeric /
              ts.total_seats::numeric, 2 ) AS fraction
  FROM (
    SELECT f.flight_id,
           f.flight_no,
           f.scheduled_departure_local,
           f.departure_city,
           f.arrival_city,
           f.aircraft_code,
           count( tf.ticket_no ) AS fact_passengers,
           ( SELECT count( s.seat_no )
               FROM seats s
               WHERE s.aircraft_code = f.aircraft_code
           ) AS total_seats
      FROM flights_v f
      JOIN ticket_flights tf ON f.flight_id = tf.flight_id
      WHERE f.status = 'Arrived'
      GROUP BY 1, 2, 3, 4, 5, 6
  ) AS ts
  JOIN aircrafts AS a ON ts.aircraft_code = a.aircraft_code
  ORDER BY ts.scheduled_departure_local;
```

Самый внутренний подзапрос — total_seats — выдает общее число мест в самолете. Этот подзапрос — коррелированный, т. к. он выполняется для каждой строки, обрабатываемой во внешнем подзапросе, т. е. для каждой модели самолета. Для подсчета числа проданных билетов мы использовали соединение представления «Рейсы» (flights_v) с таблицей «Перелеты» (ticket_flights) с последующей группировкой строк и вызовом функции count. Конечно, можно было бы вместо такого решения использовать еще один коррелированный подзапрос:

```
( SELECT count( tf.ticket_no )
    FROM ticket_flights tf
    WHERE tf.flight_id = f.flight_id
) AS fact_passengers
```

В таком случае уже не потребовалось бы выполнять соединение представления flights_v с таблицей ticket_flights и группировать строки, достаточно было бы сделать так:

```
  FROM flights_v
  WHERE f.status = 'Arrived'
) AS ts JOIN aircrafts AS a
...
```

Внешний запрос вместо кода самолета выводит наименование модели, выбирает остальные столбцы из подзапроса без изменений и дополнительно производит вычисление степени заполнения самолета пассажирами, а также сортирует результирующие строки.

```
--[ RECORD 1 ]-------------+-------------------------
 flight_id                | 28205
 flight_no                | PG0032
 scheduled_departure_local | 2016-09-13 08:00:00
 departure_city           | Пенза
 arrival_city             | Москва
 model                    | Cessna 208 Caravan
 fact_passengers          | 2
 total_seats              | 12
 fraction                 | 0.17
--[ RECORD 2 ]-------------+-------------------------
 flight_id                | 9467
 flight_no                | PG0360
 scheduled_departure_local | 2016-09-13 08:00:00
 departure_city           | Санкт-Петербург
 arrival_city             | Оренбург
 model                    | Bombardier CRJ-200
 fact_passengers          | 6
 total_seats              | 50
 fraction                 | 0.12
--[ RECORD 3 ]-------------+-------------------------
 flight_id                | 7130
 flight_no                | PG0591
 scheduled_departure_local | 2016-09-13 08:00:00
 departure_city           | Москва
 arrival_city             | Томск
 model                    | Sukhoi SuperJet-100
 fact_passengers          | 25
 total_seats              | 97
 fraction                 | 0.26
...
```

Рассмотренный сложный запрос можно сделать более наглядным за счет выделения подзапроса в отдельную конструкцию, которая называется **общее табличное выражение (Common Table Expression — CTE)**.

```
WITH ts AS
( SELECT f.flight_id,
         f.flight_no,
         f.scheduled_departure_local,
         f.departure_city,
```

```
        f.arrival_city,
        f.aircraft_code,
        count( tf.ticket_no ) AS fact_passengers,
        ( SELECT count( s.seat_no )
            FROM seats s
            WHERE s.aircraft_code = f.aircraft_code
        ) AS total_seats
    FROM flights_v f
    JOIN ticket_flights tf ON f.flight_id = tf.flight_id
    WHERE f.status = 'Arrived'
    GROUP BY 1, 2, 3, 4, 5, 6
)
SELECT ts.flight_id,
       ts.flight_no,
       ts.scheduled_departure_local,
       ts.departure_city,
       ts.arrival_city,
       a.model,
       ts.fact_passengers,
       ts.total_seats,
       round( ts.fact_passengers::numeric /
              ts.total_seats::numeric, 2 ) AS fraction
  FROM ts
  JOIN aircrafts AS a ON ts.aircraft_code = a.aircraft_code
  ORDER BY ts.scheduled_departure_local;
```

Конструкция WITH ts AS (...) и представляет собой общее табличное выражение (CTE). Такие конструкции удобны тем, что позволяют упростить основной запрос, сделать его менее громоздким. В общем табличном выражении может присутствовать больше одного подзапроса. Каждый подзапрос формирует временную таблицу с указанным именем. Если имена столбцов этой таблицы не заданы явным образом в виде списка, тогда они определяются на основе списка столбцов в предложении SELECT. В нашем примере это будет именно так. Теперь мы можем в главном запросе обращаться к временной таблице ts так, как если бы она существовала постоянно. Но важно учитывать, что временная таблица, создаваемая в общем табличном выражении, существует только во время выполнения запроса.

В этой главе мы уже решали задачу распределения сумм бронирований по диапазонам с шагом в сто тысяч рублей. Тогда мы использовали предложение VALUES для формирования виртуальной таблицы. Можно решить эту задачу более рациональным способом с использованием конструкции WITH ... AS (...).

Сначала покажем, как можно сформировать диапазоны сумм бронирований с помощью **рекурсивного общего табличного выражения**:

```
WITH RECURSIVE ranges ( min_sum, max_sum ) AS
  ( VALUES ( 0, 100000 )
    UNION ALL
    SELECT min_sum + 100000, max_sum + 100000
      FROM ranges
      WHERE max_sum <
       ( SELECT max( total_amount ) FROM bookings )
  )
SELECT * FROM ranges;
```

В этом примере мы явно указали имена столбцов для временной таблицы ranges — это min_sum и max_sum. Рекурсивный алгоритм работает следующим образом:

– сначала выполняется предложение VALUES (0, 100000) и результат записывается во временную область памяти;

– затем к этой временной области памяти применяется запрос

```
SELECT min_sum + 100000, max_sum + 100000
...
```

в результате его выполнения формируется только одна строка, поскольку в исходном предложении VALUES была сформирована только одна строка и только одна строка была помещена во временную область памяти;

– вновь сформированная строка вместе с исходной строкой помещаются в другую временную область, в которой происходит накапливание результирующих строк;

– к той строке, которая была на предыдущем шаге сформирована с помощью команды SELECT, опять применяется эта же команда и т. д.;

– работа завершится, когда перестанет выполняться условие

```
max_sum < ( SELECT max( total_amount ) FROM bookings )
```

Важную роль в этом процессе играет предложение UNION ALL, благодаря которому происходит объединение сформированных строк в единую таблицу. Поскольку в нашем примере в рекурсивном алгоритме участвует только одна строка, то строк-дубликатов не возникает, поэтому мы используем не UNION, а UNION ALL. При использовании предложения UNION выполняется устранение строк-дубликатов, но в данном случае необходимости в выполнении этой операции нет, следовательно, целесообразно использовать именно UNION ALL.

Получим такую таблицу:

```
 min_sum | max_sum
---------+--------
       0 |  100000
  100000 |  200000
  200000 |  300000
  300000 |  400000
  400000 |  500000
  500000 |  600000
  600000 |  700000
  700000 |  800000
  800000 |  900000
  900000 | 1000000
 1000000 | 1100000
 1100000 | 1200000
 1200000 | 1300000
(13 строк)
```

Здесь в предложении WHERE используется скалярный подзапрос. С результатом его выполнения сравнивается значение столбца max_sum:

```
WHERE max_sum < ( SELECT max( total_amount ) FROM bookings )
```

Теперь давайте скомбинируем рекурсивное общее табличное выражение с выборкой из таблицы bookings:

```
WITH RECURSIVE ranges ( min_sum, max_sum ) AS
  ( VALUES( 0, 100000 )
    UNION ALL
    SELECT min_sum + 100000, max_sum + 100000
      FROM ranges
      WHERE max_sum <
        ( SELECT max( total_amount ) FROM bookings )
  )
SELECT r.min_sum, r.max_sum, count( b.* )
  FROM bookings b
  RIGHT OUTER JOIN ranges r
    ON b.total_amount >= r.min_sum
    AND b.total_amount < r.max_sum
  GROUP BY r.min_sum, r.max_sum
  ORDER BY r.min_sum;
```

```
 min_sum | max_sum | count
---------+---------+--------
       0 |  100000 | 198314
  100000 |  200000 |  46943
  200000 |  300000 |  11916
  300000 |  400000 |   3260
  400000 |  500000 |   1357
  500000 |  600000 |    681
  600000 |  700000 |    222
  700000 |  800000 |     55
  800000 |  900000 |     24
  900000 | 1000000 |     11
 1000000 | 1100000 |      4
 1100000 | 1200000 |      0
 1200000 | 1300000 |      1
(13 строк)
```

Обратите внимание, что для диапазона от 1100 до 1200 тысяч рублей значение числа бронирований равно нулю. Для того чтобы была выведена строка с нулевым значением столбца count, мы использовали внешнее соединение.

В заключение рассмотрим команду для создания материализованного представления «Маршруты» (routes), которое было описано в главе 5. Но тогда мы не стали рассматривать эту команду, т. к. еще не ознакомились с подзапросами, которые в ней используются.

| Описание атрибута | Имя атрибута | Тип PostgreSQL |
|---|---|---|
| Номер рейса | flight_no | char(6) |
| Код аэропорта отправления | departure_airport | char(3) |
| Название аэропорта отправления | departure_airport_name | text |
| Город отправления | departure_city | text |
| Код аэропорта прибытия | arrival_airport | char(3) |
| Название аэропорта прибытия | arrival_airport_name | text |
| Город прибытия | arrival_city | text |
| Код самолета, IATA | aircraft_code | char(3) |
| Продолжительность полета | duration | interval |
| Дни недели, когда выполняются рейсы | days_of_week | integer[] |

Эта команда выглядит так:

```
CREATE MATERIALIZED VIEW routes AS
  WITH f3 AS
    ( SELECT f2.flight_no,
             f2.departure_airport,
             f2.arrival_airport,
             f2.aircraft_code,
             f2.duration,
             array_agg( f2.days_of_week ) AS days_of_week
        FROM ( SELECT f1.flight_no,
                      f1.departure_airport,
                      f1.arrival_airport,
                      f1.aircraft_code,
                      f1.duration,
                      f1.days_of_week
                 FROM ( SELECT flights.flight_no,
                               flights.departure_airport,
                               flights.arrival_airport,
                               flights.aircraft_code,
                             ( flights.scheduled_arrival -
                               flights.scheduled_departure
                             ) AS duration,
                             ( to_char( flights.scheduled_departure,
                               'ID'::text ))::integer AS days_of_week
                          FROM flights
                 ) f1
                 GROUP BY f1.flight_no, f1.departure_airport,
                          f1.arrival_airport, f1.aircraft_code,
                          f1.duration, f1.days_of_week
                 ORDER BY f1.flight_no, f1.departure_airport,
                          f1.arrival_airport, f1.aircraft_code,
                          f1.duration, f1.days_of_week
        ) f2
        GROUP BY f2.flight_no, f2.departure_airport,
                 f2.arrival_airport, f2.aircraft_code, f2.duration
    )
  SELECT f3.flight_no,
         f3.departure_airport,
         dep.airport_name AS departure_airport_name,
         dep.city AS departure_city,
```

```
    f3.arrival_airport,
    arr.airport_name AS arrival_airport_name,
    arr.city AS arrival_city,
    f3.aircraft_code,
    f3.duration,
    f3.days_of_week
FROM f3,
    airports dep,
    airports arr
WHERE f3.departure_airport = dep.airport_code
  AND f3.arrival_airport   = arr.airport_code;
```

Начнем ознакомление с запросом с его верхней части. Здесь мы видим конструкцию WITH f3 AS (...), т. е. общее табличное выражение. В результате его выполнения будет сформирована временная таблица f3. Запрос, который ее формирует, содержит в предложении FROM подзапрос, формирующий временную таблицу f2. А этот подзапрос, в свою очередь, также содержит в предложении FROM подзапрос, формирующий временную таблицу f1. Таким образом, в этой команде используется вложенный подзапрос.

Во вложенном подзапросе используется функция to_char. Второй ее параметр — «ID» — указывает на то, что из значения даты/времени вылета будет извлечен номер дня недели. При этом нумерация дней недели соответствует стандарту ISO 8601: понедельник — 1, воскресенье — 7. Поскольку номер дня недели представлен в виде символьной строки, он преобразуется в тип данных integer. Таким образом, вложенный подзапрос вычисляет плановую длительность полета (столбец duration) и извлекает номер дня недели из даты/времени вылета по расписанию (столбец days_of_week).

Подзапрос следующего, более высокого уровня, получив результат вложенного подзапроса, просто группирует строки, готовя столбец days_of_week к объединению отдельных номеров дней недели в массивы целых чисел. При этом в предложение GROUP BY включен столбец days_of_week, чтобы заменить дубликаты дней недели одним значением. Ведь таблица flights содержит расписание рейсов на длительный период. Поэтому рейс, который отправляется, скажем, по вторникам, появится в этом расписании несколько раз, следовательно, день недели с номером 2 также появится в столбце days_of_week для этого номера рейса несколько раз. В результате, если не прибегнуть к группировке по этому столбцу, то при формировании массива дней недели в этом массиве будут многократные вхождения каждого дня недели, когда этот рейс летает. В этом подзапросе присутствует и предложение ORDER BY, в которое включен столбец days_of_week. Это необходимо для того, чтобы агрегатная функция array_agg собрала номера дней недели в массив в возрастающем порядке этих номеров.

Во внешнем запросе вызывается функция array_agg, которая агрегирует номера дней недели, содержащиеся в сгруппированных строках, в массивы целых чисел. На этом работа конструкции WITH f3 AS (...) завершается. В результате вместо нескольких строк в таблице flights, соответствующих вылетам конкретного рейса в различные дни недели, формируется одна строка в представлении routes, в этой строке все дни недели, в которые выполняется конкретный рейс, собраны в массив целых чисел.

И, наконец, главный запрос выполняет соединение временной таблицы f3 с таблицей «Аэропорты» (airports), причем, дважды. Это нужно потому, что в таблице f3 есть

столбец f3.departure_airport (аэропорт отправления) и столбец f3.arrival_airport (аэропорт прибытия), для каждого из них нужно выбрать наименование аэропорта и наименование города из таблицы airports. О том, как нужно рассуждать при двукратном использовании одной и той же таблицы в соединении, мы уже говорили ранее в разделе 5.4 «Представления».

## Контрольные вопросы и задания

1. В документации сказано, что служебный символ «%» в шаблоне оператора LIKE соответствует любой последовательности символов, в том числе и пустой последовательности, однако ничего не сказано насчет правил обработки пробелов.

   В таблице «Билеты» (tickets) столбец passenger_name содержит имя и фамилию пассажира, записанные заглавными латинскими буквами и разделенные одним пробелом.

   Выясните правила обработки пробелов самостоятельно, выполнив следующие команды и сравнив полученные результаты:

   ```
   SELECT count( * ) FROM tickets;
   SELECT count( * ) FROM tickets WHERE passenger_name LIKE '% %';
   SELECT count( * ) FROM tickets WHERE passenger_name LIKE '% % %';
   SELECT count( * ) FROM tickets WHERE passenger_name LIKE '% %%';
   ```

2. Этот запрос выбирает из таблицы «Билеты» (tickets) всех пассажиров с именами, состоящими из трех букв (в шаблоне присутствует три символа «_»):

   ```
   SELECT passenger_name
     FROM tickets
     WHERE passenger_name LIKE '___ %';
   ```

   Предложите шаблон поиска в операторе LIKE для выбора из этой таблицы всех пассажиров с фамилиями, состоящими из пяти букв.

3. В разделе 9.7.2 «Регулярные выражения SIMILAR TO» рассматривается оператор SIMILAR TO. Он работает аналогично оператору LIKE, но использует шаблоны, соответствующие определению регулярных выражений, приведенному в стандарте SQL. Регулярные выражения SQL представляют собой комбинацию синтаксиса LIKE с синтаксисом обычных регулярных выражений. Самостоятельно ознакомьтесь с оператором SIMILAR TO.

4. В разделе 9.2 «Функция и операторы сравнения» представлены различные предикаты сравнения, кроме предиката BETWEEN, рассмотренного в этой главе. Самостоятельно ознакомьтесь с ними.

5. В разделе 9.17 «Условные выражения» представлены условные выражения, которые поддерживаются в PostgreSQL. В тексте главы была рассмотрена конструкция CASE. Самостоятельно ознакомьтесь с функциями COALESCE, NULLIF, GREATEST и LEAST.

6. Выясните, на каких маршрутах используются самолеты компании Boeing? В выборке вместо кода модели должно выводиться ее наименование, например, вместо кода 733 должно быть Boeing 737-300.

   Указание: можно воспользоваться соединением представления «Маршруты» (routes) и таблицы «Самолеты» (aircrafts).

7. Самые крупные самолеты в нашей авиакомпании — это Boeing 777-300. Выяснить, между какими парами городов они летают, поможет запрос:

```
SELECT DISTINCT departure_city, arrival_city
  FROM routes r
  JOIN aircrafts a ON r.aircraft_code = a.aircraft_code
  WHERE a.model = 'Boeing 777-300'
  ORDER BY 1;
```

```
 departure_city | arrival_city
----------------+--------------
 Екатеринбург   | Москва
 Москва         | Екатеринбург
 Москва         | Новосибирск
 Москва         | Пермь
 Москва         | Сочи
 Новосибирск    | Москва
 Пермь          | Москва
 Сочи           | Москва
(8 строк)
```

   К сожалению, в этой выборке информация дублируется. Пары городов приведены по два раза: для рейса «туда» и для рейса «обратно». Модифицируйте запрос таким образом, чтобы каждая пара городов была выведена только один раз:

```
 departure_city | arrival_city
----------------+--------------
 Москва         | Екатеринбург
 Новосибирск    | Москва
 Пермь          | Москва
 Сочи           | Москва
(4 строки)
```

8. В тексте главы мы рассматривали различные примеры использования левого и правого внешних соединений: LEFT OUTER JOIN и RIGHT OUTER JOIN. Напишите запрос, в котором использовалось бы полное внешнее соединение — FULL OUTER JOIN.

9. Для получения ответа на вопрос, сколько рейсов выполняется из Москвы в Санкт-Петербург, можно написать совсем простой запрос:

```
SELECT count( * )
  FROM routes
  WHERE departure_city = 'Москва'
    AND arrival_city   = 'Санкт-Петербург';
```

```
 count
-------
    12
(1 строка)
```

А с помощью какого запроса можно получить результат в таком виде?

```
 departure_city |  arrival_city   | count
----------------+-----------------+-------
 Москва         | Санкт-Петербург |    12
(1 строка)
```

10. Выяснить, сколько различных рейсов выполняется из каждого города, без учета частоты рейсов в неделю, можно с помощью обращения к представлению «Маршруты» (routes):

```
SELECT departure_city, count( * )
  FROM routes
  GROUP BY departure_city
  ORDER BY count DESC;
```

```
     departure_city       | count
--------------------------+-------
 Москва                   |   154
 Санкт-Петербург          |   35
 Новосибирск              |    19
 Екатеринбург             |    15
 Ростов-на-Дону           |    14
 Сочи                     |    14
 Красноярск               |    13
 Ульяновск                |    11
 . . .
 Благовещенск             |     1
 Братск                   |     1
(101 строка)
```

Модифицируйте этот запрос так, чтобы он выводил число направлений, по которым летают самолеты из каждого города. Например, из Москвы в Санкт-Петербург летает несколько различных рейсов, но все эти рейсы относятся к одному направлению.

Указание: нужно передать параметр в функцию count.

11. В материализованном представлении «Маршруты» (routes) есть столбец days_of_week, который содержит списки (массивы) номеров дней недели, когда выполняется каждый рейс.

Для оптимизации расписания вылетов из Москвы нужно выявить пять городов, в которые из столицы отправляется наибольшее число ежедневных рейсов (маршрутов). Строки в выборке следует расположить в убывающем порядке числа выполняемых рейсов.

Указание. Воспользуйтесь функцией array_length.

12.* Предположим, что служба материального снабжения нашей авиакомпании запросила информацию о числе рейсов, выполняющихся из Москвы в каждый день недели.

Результат можно получить путем выполнения семи аналогичных запросов: по одному для каждого дня недели. Начнем с понедельника:

```
SELECT 'Понедельник' AS day_of_week, count( * ) AS num_flights
  FROM routes
  WHERE departure_city = 'Москва'
    AND days_of_week @> '{ 1 }'::integer[];
```

В этом запросе используется оператор «@>», который проверяет, содержатся ли все элементы массива, стоящего справа от него, в том массиве, который находится слева. В правом массиве всего один элемент — номер интересующего нас дня недели.

```
 day_of_week | num_flights
-------------+-------------
 Понедельник |         131
(1 строка)
```

Запрос для вторника отличается лишь названием дня недели и его номером в массиве.

```
 SELECT 'Вторник' AS day_of_week, count( * ) AS num_flights
   FROM routes
   WHERE departure_city = 'Москва'
     AND days_of_week @> '{ 2 }'::integer[];
```

```
 day_of_week | num_flights
-------------+-------------
 Вторник     |         134
(1 строка)
```

Нужно выполнить еще пять аналогичных команд, чтобы получить результаты для всех дней недели. Очевидно, что это нерациональный способ.

Получить требуемый результат можно с помощью одного запроса:

```
SELECT unnest( days_of_week ) AS day_of_week,
       count( * ) AS num_flights
  FROM routes
  WHERE departure_city = 'Москва'
  GROUP BY day_of_week
  ORDER BY day_of_week;
```

```
 day_of_week | num_flights
-------------+-------------
           1 |         131
           2 |         134
           3 |         126
           4 |         136
           5 |         124
           6 |         133
           7 |         124
(7 строк)
```

**Задание 1.** Самостоятельно разберитесь, как работает приведенный запрос. Выясните, что делает функция unnest. Для того чтобы найти ее описание, можно воспользоваться теми разделами документации, которые были указаны в тексте главы 4. Однако можно воспользоваться и предметным указателем (Index), ссылка на который находится в самом низу оглавления документации.

В качестве вспомогательного запроса, проясняющего работу функции unnest, можно выполнить следующий:

```
SELECT flight_no, unnest( days_of_week ) AS day_of_week
  FROM routes
  WHERE departure_city = 'Москва'
  ORDER BY flight_no;
```

**Задание 2.** Использование номеров дней недели в предыдущей выборке не должно вызывать затруднений. Но все-таки предположим, что нас попросили модифицировать запрос, чтобы результат выводился в таком виде:

```
 name_of_day | num_flights
-------------+-------------
 Пн.         |         131
 Вт.         |         134
 Ср.         |         126
 Чт.         |         136
 Пт.         |         124
 Сб.         |         133
 Вс.         |         124
(7 строк)
```

Покажем одно из возможных решений задачи. Оно основано на использовании специальной табличной функции unnest в предложении FROM. Подробно об этом написано в документации в разделе 7.2.1.4 «Табличные функции». Функция unnest может принимать любое число параметров-массивов, а возвращает набор строк, которые могут использоваться в запросах, как обычные таблицы. В этих наборах строк столбцы формируются из значений, содержащихся в массивах.

```
SELECT dw.name_of_day, count( * ) AS num_flights
  FROM (
    SELECT unnest( days_of_week ) AS num_of_day
      FROM routes
      WHERE departure_city = 'Москва'
    ) AS r,
    unnest( '{ 1, 2, 3, 4, 5, 6, 7 }'::integer[],
     '{ "Пн.", "Вт.", "Ср.", "Чт.", "Пт.", "Сб.", "Вс."}'::text[]
    ) AS dw( num_of_day, name_of_day )
  WHERE    r.num_of_day = dw.num_of_day
  GROUP BY r.num_of_day,  dw.name_of_day
  ORDER BY r.num_of_day;
```

Этот запрос можно упростить, воспользовавшись предложением WITH ORDINALITY. Оно позволяет в нашем примере избавиться от массива целых чисел, обозначающих дни недели, поскольку автоматически формируется

столбец целых чисел, нумерующих строки результирующего набора. По умолчанию этот столбец называется ordinality. Это имя можно использовать в запросе.

Самостоятельно модифицируйте запрос с применением предложения WITH ORDINALITY.

13. Ответить на вопрос о том, каковы максимальные и минимальные цены билетов на все направления, может такой запрос:

```
SELECT f.departure_city, f.arrival_city,
    max( tf.amount ), min( tf.amount )
  FROM flights_v f
  JOIN ticket_flights tf ON f.flight_id = tf.flight_id
  GROUP BY 1, 2
  ORDER BY 1, 2;
```

```
   departure_city    |     arrival_city     |    max    |   min
---------------------+----------------------+-----------+----------
 Абакан              | Москва               | 101000.00 | 33700.00
 Абакан              | Новосибирск          |   5800.00 |  5800.00
 Абакан              | Томск                |   4900.00 |  4900.00
 Анадырь             | Москва               | 185300.00 | 61800.00
 Анадырь             | Хабаровск            |  92200.00 | 30700.00
 ...
 Якутск              | Мирный               |   8900.00 |  8100.00
 Якутск              | Санкт-Петербург      | 145300.00 | 48400.00
(367 строк)
```

А как выявить те направления, на которые не было продано ни одного билета? Один из вариантов решения такой: если на рейсы, отправляющиеся по какому-то направлению, не было продано ни одного билета, то максимальная и минимальная цены будут равны NULL. Нужно получить выборку в таком виде:

```
   departure_city    |    arrival_city      |    max    |    min
---------------------+----------------------+-----------+----------
 Абакан              | Архангельск          |           |
 Абакан              | Грозный              |           |
 Абакан              | Кызыл                |           |
 Абакан              | Москва               | 101000.00 | 33700.00
 Абакан              | Новосибирск          |   5800.00 |  5800.00
...
```

Модифицируйте запрос, приведенный выше.

14. Предположим, что маркетологи нашей авиакомпании хотят знать, как часто встречаются различные имена среди пассажиров? Получить распределение частот имен пассажиров в таблице «Билеты» (tickets) поможет такой запрос:

```
SELECT left( passenger_name, strpos( passenger_name, ' ' ) - 1 )
       AS firstname, count( * )
  FROM tickets
  GROUP BY 1
  ORDER BY 2 DESC;
```

```
  firstname | count
------------+-------
  ALEKSANDR | 20328
  SERGEY    | 15133
  VLADIMIR  | 12806
  TATYANA   | 12058
  ELENA     | 11291
  OLGA      |  9998
 ...
  MAGOMED   |    14
  ASKAR     |    13
  RASUL     |    11
(363 строки)
```

Напишите запрос для ответа на аналогичный вопрос насчет распределения частот фамилий пассажиров.

Подробные сведения о других функциях для работы со строковыми данными приведены в документации в разделе 9.4 «Строковые функции и операторы».

15.* В тексте главы были кратко рассмотрены оконные функции. Самостоятельно ознакомьтесь с разделами документации, которые рекомендуется изучить для более детального ознакомления с этим классом функций.

Подумайте, в какой ситуации, связанной с базой данных «Авиаперевозки», было бы полезно применить оконные функции, и напишите запрос.

16.* Вместе с агрегатными функциями может использоваться предложение FILTER. Самостоятельно ознакомьтесь с этой темой, обратившись к разделу документации 4.2.7 «Агрегатные выражения». Напишите запрос с использованием предложения FILTER с агрегатной функцией.

17. В тексте главы в разделе 6.4 мы рассмотрели два способа получения ответа на вопрос: как распределяются места с разными классами обслуживания в самолетах всех типов?

А с помощью какого запроса можно получить результат в таком виде:

```
 aircraft_code |       model         | fare_conditions | count
---------------+---------------------+-----------------+-------
 319           | Airbus A319-100     | Business        |    20
 319           | Airbus A319-100     | Economy         |    96
...
 CR2           | Bombardier CRJ-200  | Economy         |    50
 SU9           | Sukhoi SuperJet-100 | Business        |    12
 SU9           | Sukhoi SuperJet-100 | Economy         |    85
(17 строк)
```

18. В разделе 6.2 «Соединения» мы находили ответ на вопрос: сколько маршрутов обслуживают самолеты каждого типа? Но для повышения наглядности получаемых результатов необходимо еще рассчитывать относительные величины, т. е. доли от общего числа маршрутов.

Вот что требуется получить:

```
 a_code |         model        | r_code | num_routes | fraction
--------+----------------------+--------+------------+----------
 CR2    | Bombardier CRJ-200   | CR2    |        232 |    0.327
 CN1    | Cessna 208 Caravan   | CN1    |        170 |    0.239
 ...
 773    | Boeing 777-300       | 773    |         10 |    0.014
 320    | Airbus A320-200      |        |          0 |    0.000
(9 строк)
```

19.* В разделе 6.4 «Подзапросы» мы использовали рекурсивный алгоритм в общем табличном выражении. Изучите этот пример, чтобы лучше понять работу рекурсивного алгоритма:

```
WITH RECURSIVE ranges ( min_sum, max_sum ) AS
  ( VALUES( 0,      100000 ),
          ( 100000, 200000 ),
          ( 200000, 300000 )
    UNION ALL
    SELECT min_sum + 100000, max_sum + 100000
      FROM ranges
      WHERE max_sum <
        ( SELECT max( total_amount ) FROM bookings )
  )
SELECT * FROM ranges;
```

```
 min_sum | max_sum
---------+---------
       0 |  100000       исходные строки
  100000 |  200000
  200000 |  300000

  100000 |  200000       результат первой итерации
  200000 |  300000
  300000 |  400000

  200000 |  300000       результат второй итерации
  300000 |  400000
  400000 |  500000

  300000 |  400000
  400000 |  500000
  500000 |  600000
  ...
 1000000 | 1100000       результат (n-3)-й итерации
 1100000 | 1200000
 1200000 | 1300000

 1100000 | 1200000       результат (n-2)-й итерации
 1200000 | 1300000

 1200000 | 1300000       результат (n-1)-й итерации (предпоследней)
(36 строк)
```

Здесь мы с помощью предложения VALUES специально создали виртуальную таблицу из трех строк, хотя для получения требуемого результата достаточно

только одной строки (0, 100000). Еще важно то, что предложение UNION ALL не удаляет строки-дубликаты, поэтому мы можем видеть весь рекурсивный процесс порождения новых строк.

При рекурсивном выполнении запроса

```
SELECT min_sum + 100000, max_sum + 100000 ...
```

каждый раз выполняется проверка в условии WHERE. И на $(n-2)$-й итерации это условие отсеивает одну строку, т. к. после $(n-3)$-й итерации значение атрибута max_sum в третьей строке было равно 1 300 000. Ведь запрос

```
SELECT max( total_amount ) FROM bookings;
```

выдаст значение

```
    max
------------
 1204500.00
(1 строка)
```

Таким образом, после $(n-2)$-й итерации во временной области остается всего две строки, после $(n-1)$-й итерации во временной области остается только одна строка. Заключительная итерация уже не добавляет строк в результирующую таблицу, поскольку единственная строка, поданная на вход команде SELECT, будет отклонена условием WHERE. Работа алгоритма завершается.

**Задание 1.** Модифицируйте запрос, добавив в него столбец level (можно назвать его и iteration). Этот столбец должен содержать номер текущей итерации, поэтому нужно увеличивать его значение на единицу на каждом шаге. Не забудьте задать начальное значение для добавленного столбца в предложении VALUES.

**Задание 2.** Для завершения экспериментов замените UNION ALL на UNION и выполните запрос. Сравните этот результат с предыдущим, когда мы использовали UNION ALL.

20.* В тексте главы есть такой запрос, вычисляющий распределение сумм бронирований по диапазонам в сто тысяч рублей:

```
WITH RECURSIVE ranges ( min_sum, max_sum ) AS
( VALUES( 0, 100000 )
  UNION ALL
  SELECT min_sum + 100000, max_sum + 100000
    FROM ranges
    WHERE max_sum <
      ( SELECT max( total_amount ) FROM bookings )
)
SELECT r.min_sum, r.max_sum, count( b.* )
  FROM bookings b
  RIGHT OUTER JOIN ranges r
    ON  b.total_amount >= r.min_sum
    AND b.total_amount <  r.max_sum
  GROUP BY r.min_sum, r.max_sum
  ORDER BY r.min_sum;
```

Как вы думаете, почему функция count получает в качестве параметра выражение «b.*», а не просто «*»? Что изменится, если оставить только «*», и почему?

21. В тексте главы был приведен запрос, выводящий список городов, в которые нет рейсов из Москвы.

```
SELECT DISTINCT a.city
  FROM airports a
  WHERE NOT EXISTS (
    SELECT * FROM routes r
      WHERE r.departure_city = 'Москва'
        AND r.arrival_city = a.city
    )
    AND a.city <> 'Москва'
  ORDER BY city;
```

Можно предложить другой вариант, в котором используется одна из операций над множествами строк: объединение, пересечение или разность. Вместо знака «?» поставьте в запросе нужное ключевое слово — UNION, INTERSECT или EXCEPT — и обоснуйте ваше решение.

```
SELECT city
  FROM airports
  WHERE city <> 'Москва'
?
SELECT arrival_city
  FROM routes
  WHERE departure_city = 'Москва'
ORDER BY city;
```

22. В тексте главы мы рассматривали такой запрос: получить перечень аэропортов в тех городах, в которых больше одного аэропорта.

```
SELECT aa.city, aa.airport_code, aa.airport_name
  FROM (
    SELECT city, count( * )
      FROM airports
      GROUP BY city
      HAVING count( * ) > 1
    ) AS a
  JOIN airports AS aa ON a.city = aa.city
  ORDER BY aa.city, aa.airport_name;
```

Как вы думаете, обязательно ли наличие функции count в подзапросе в предложении SELECT или можно написать просто

```
...
FROM ( SELECT city FROM airports
...
```

Сначала попробуйте дать ответ теоретически, а потом проверьте вашу гипотезу на компьютере.

23. Предположим, что департамент развития нашей авиакомпании задался вопросом: каким будет общее число различных маршрутов, которые теоретически можно проложить между всеми городами?

    Если в каком-то городе имеется более одного аэропорта, то это учитывать не будем, т. е. маршрутом будем считать путь между *городами*, а не между *аэропортами*. Здесь мы используем соединение таблицы с самой собой на основе неравенства значений атрибутов.

```
SELECT count( * )
  FROM ( SELECT DISTINCT city FROM airports ) AS a1
  JOIN ( SELECT DISTINCT city FROM airports ) AS a2
     ON a1.city <> a2.city;
```

```
 count
-------
 10100
(1 строка)
```

    **Задание.** Перепишите этот запрос с использованием общего табличного выражения.

24. В тексте главы мы рассмотрели использование подзапросов в предикатах EXISTS и IN. Существуют также предикаты многократного сравнения ANY и ALL. Они представлены в документации в разделе 9.22 «Выражения подзапросов». Самостоятельно ознакомьтесь с этими предикатами и напишите несколько запросов с их применением.

    Предикаты ANY и ALL имеют некоторую связь с предикатом IN. В частности, использование IN эквивалентно использованию конструкции = ANY, а использование NOT IN эквивалентно использованию конструкции <> ALL.

    Пример двух эквивалентных запросов, выбирающих аэропорты в часовых поясах «Asia/Novokuznetsk» и «Asia/Krasnoyarsk»:

```
SELECT * FROM airports
  WHERE timezone IN
     ( 'Asia/Novokuznetsk', 'Asia/Krasnoyarsk' );
```

```
SELECT * FROM airports
  WHERE timezone = ANY (
    VALUES ( 'Asia/Novokuznetsk' ),
           ( 'Asia/Krasnoyarsk'  )
  );
```

    Еще один пример. В тексте главы мы рассматривали запрос, подсчитывающий количество маршрутов, проложенных из самых восточных аэропортов.

```
SELECT departure_city, count( * )
  FROM routes
  GROUP BY departure_city
  HAVING departure_city IN (
    SELECT city
      FROM airports
      WHERE longitude > 150
```

```
  )
  ORDER BY count DESC;
```

В этом запросе можно заменить IN на ANY таким образом:

```
...
HAVING departure_city = ANY ( SELECT city
...
```

25.* При планировании новых маршрутов и оценке экономической эффективности уже существующих может потребоваться информация о том, какова усредненная степень заполнения самолетов на всех направлениях.

Будем учитывать только уже прибывшие рейсы.

```
WITH tickets_seats AS
  ( SELECT f.flight_id, f.flight_no, f.departure_city,
           f.arrival_city, f.aircraft_code,
           count( tf.ticket_no ) AS fact_passengers,
           ( SELECT count( s.seat_no )
               FROM seats s
               WHERE s.aircraft_code = f.aircraft_code
           ) AS total_seats
      FROM flights_v f
      JOIN ticket_flights tf ON f.flight_id = tf.flight_id
      WHERE f.status = 'Arrived'
      GROUP BY 1, 2, 3, 4, 5
  )
SELECT ts.departure_city, ts.arrival_city,
    sum( ts.fact_passengers ) AS sum_fact_passengers,
    sum( ts.total_seats ) AS sum_total_seats,
    round( sum( ts.fact_passengers )::numeric /
           sum( ts.total_seats )::numeric, 2 ) AS fraction
  FROM tickets_seats ts
  GROUP BY ts.departure_city, ts.arrival_city
  ORDER BY ts.departure_city;
```

```
--[ RECORD 1 ]-------+------------------------
 departure_city      | Абакан
 arrival_city        | Москва
 sum_fact_passengers | 466
 sum_total_seats     | 1044
 fraction            | 0.45
--[ RECORD 2 ]-------+------------------------
 departure_city      | Абакан
 arrival_city        | Новосибирск
 sum_fact_passengers | 217
 sum_total_seats     | 348
 fraction            | 0.62
--[ RECORD 3 ]-------+------------------------
 departure_city      | Абакан
 arrival_city        | Томск
 sum_fact_passengers | 258
 sum_total_seats     | 360
 fraction            | 0.72
```

```
...
--[ RECORD 361 ]-----+------------------------
 departure_city      | Якутск
 arrival_city        | Санкт-Петербург
 sum_fact_passengers | 352
 sum_total_seats     | 3596
 fraction            | 0.10
```

Для того чтобы лучше уяснить, как работает запрос в целом, вычлените из него отдельные подзапросы и выполните их, посмотрите, что они выводят.

Как вы считаете, равносильно ли в данном запросе

```
SELECT count( s.seat_no )
```

и

```
SELECT count( s.* )
```

Почему?

Задание. Модифицируйте этот запрос, чтобы он выводил те же отчетные данные, но с учетом классов обслуживания, т. е. Business, Comfort и Economy.

26.* Предположим, что некая контролирующая организация потребовала информацию о размещении пассажиров одного из рейсов Кемерово — Москва в салоне самолета. Для определенности выберем конкретный рейс из тех рейсов, которые уже прибыли на момент времени, соответствующий текущему моменту. Текущий момент времени в базе данных «Авиаперевозки» определяется с помощью функции bookings.now.

Выполним запрос

```
SELECT *
  FROM flights_v
  WHERE departure_city = 'Кемерово'
    AND arrival_city   = 'Москва'
    AND actual_arrival < bookings.now();
```

Выберем для дальнейшей работы рейс, у которого значения атрибутов flight_id — 27584, aircraft_code — SU9.

Получим список пассажиров этого рейса с местами, которые им были назначены в салоне самолета.

```
SELECT t.passenger_name, b.seat_no
  FROM (
    ticket_flights tf
    JOIN tickets t ON tf.ticket_no = t.ticket_no
  )
  JOIN boarding_passes b
    ON  tf.ticket_no = b.ticket_no
    AND tf.flight_id = b.flight_id
  WHERE tf.flight_id = 27584
  ORDER BY t.passenger_name;
```

```
 passenger_name      | seat_no
---------------------+---------
 ALEKSANDR ABRAMOV   | 1A
 ALEKSANDR GRIGOREV  | 5C
 ALEKSANDR SERGEEV   | 6F
 ALEKSEY FEDOROV     | 11D
 ALEKSEY MELNIKOV    | 18A
...
 VLADIMIR POPOV      | 11A
 YAROSLAV KUZMIN     | 18F
 YURIY ZAKHAROV      | 10F
(44 строки)
```

Отсортируем строки по фамилиям пассажиров:

```
SELECT t.passenger_name,
    substr(   t.passenger_name,
      strpos( t.passenger_name, ' ' ) + 1
    ) AS lastname,
    left(     t.passenger_name,
      strpos( t.passenger_name, ' ' ) - 1
    ) AS firstname,
    b.seat_no
  FROM (
    ticket_flights tf
    JOIN tickets t ON tf.ticket_no = t.ticket_no
  )
  JOIN boarding_passes b
    ON  tf.ticket_no = b.ticket_no
    AND tf.flight_id = b.flight_id
  WHERE tf.flight_id = 27584
  ORDER BY 2, 3;
```

```
    passenger_name   | lastname  | firstname | seat_no
---------------------+-----------+-----------+---------
 ALEKSANDR ABRAMOV   | ABRAMOV   | ALEKSANDR | 1A
 NIKITA ANDREEV      | ANDREEV   | NIKITA    | 6D
 ANTONINA ANISIMOVA  | ANISIMOVA | ANTONINA  | 11F
...
 YURIY ZAKHAROV      | ZAKHAROV  | YURIY     | 10F
 ELENA ZOTOVA        | ZOTOVA    | ELENA     | 20E
(44 строки)
```

Получим список мест в салоне самолета и пассажиров, которые сидели на этих местах. При этом незанятые места также должны быть выведены (поэтому используем левое внешнее соединение ... FROM seats s LEFT OUTER JOIN ...).

```
SELECT s.seat_no, p.passenger_name
  FROM seats s
  LEFT OUTER JOIN (
    SELECT t.passenger_name, b.seat_no
      FROM (
        ticket_flights tf
        JOIN tickets t ON tf.ticket_no = t.ticket_no
      )
```

```
      JOIN boarding_passes b
        ON  tf.ticket_no = b.ticket_no
        AND tf.flight_id = b.flight_id
      WHERE tf.flight_id = 27584
  ) AS p
     ON s.seat_no = p.seat_no
  WHERE s.aircraft_code = 'SU9'
  ORDER BY s.seat_no;
```

```
 seat_no | passenger_name
---------+---------------------
 10A     |
 10C     |
 10D     | NATALYA POPOVA
 10E     |
 10F     | YURIY ZAKHAROV
 11A     | VLADIMIR POPOV
 11C     | ANNA KUZMINA
 11D     | ALEKSEY FEDOROV
 11E     |
 11F     | ANTONINA ANISIMOVA
 ...
 8F      |
 9A      | MAKSIM CHERNOV
 9C      |
 9D      | LYUDMILA IVANOVA
 9E      |
 9F      | SOFIYA KULIKOVA
(97 строк)
```

Предположим, что нас попросили отсортировать места в порядке их расположения в салоне самолета и вывести также адреса электронной почты пассажиров (у кого они были указаны при бронировании). Для выполнения второго требования воспользуемся столбцом contact_data. В нем содержатся JSON-объекты, содержащие контактные данные пассажиров. Ряд из них имеет ключ «email». Модифицированный запрос будет таким:

```
SELECT s.seat_no, p.passenger_name, p.email
  FROM seats s
  LEFT OUTER JOIN (
    SELECT t.passenger_name, b.seat_no,
           t.contact_data->'email' AS email
      FROM (
        ticket_flights tf
        JOIN tickets t ON tf.ticket_no = t.ticket_no
      )
      JOIN boarding_passes b
        ON  tf.ticket_no = b.ticket_no
        AND tf.flight_id = b.flight_id
      WHERE tf.flight_id = 27584
  ) AS p
     ON s.seat_no = p.seat_no
  WHERE s.aircraft_code = 'SU9'
  ORDER BY
```

```
   left( s.seat_no, length( s.seat_no ) - 1 )::integer,
   right( s.seat_no, 1 );
```

```
 seat_no |   passenger_name  | email
---------+-------------------+-----------------------------------
 1A      | ALEKSANDR ABRAMOV |
 1C      |                   |
 1D      | DENIS PETROV      |
 1F      | LEONID BARANOV    | "baranov.l.1967@postgrespro.ru"
 2A      |                   |
 2C      |                   |
 2D      |                   |
 2F      | FEDOR TIKHONOV    | "tikhonov_fedor_1957@postgres... "
 ...
 9F      | SOFIYA KULIKOVA   | "sofiya.kulikova_041963@postgre..."
 10A     |                   |
 10C     |                   |
 10D     | NATALYA POPOVA    | "popova.n_13031976@postgrespro.ru"
 ...
 20E     | ELENA ZOTOVA      |
 20F     | LILIYA OSIPOVA    |
(97 строк)
```

Задание. Перепишите последний запрос с использованием общего табличного выражения и добавьте столбец «Класс обслуживания» (fare_conditions).

# 7 Изменение данных

Эта глава будет посвящена операциям изменения данных: вставке новых строк в таблицы, обновлению уже существующих строк и их удалению. С простыми приемами использования команд INSERT, UPDATE и DELETE, предназначенных для выполнения указанных операций, вы уже познакомились, поэтому мы расскажем о некоторых более интересных способах применения этих команд.

## 7.1 Вставка строк в таблицы

Для работы нам потребуется создать еще две таблицы в базе данных «Авиаперевозки» (demo). Мы будем создавать их как временные таблицы, которые будут удаляться при отключении от базы данных. Использование временных таблиц позволит нам проводить эксперименты, будучи уверенными в том, что данные в постоянных таблицах модифицированы не будут, поэтому все запросы, которые вы выполняли ранее, будут работать так, как и работали.

Итак, создадим две копии таблицы «Самолеты» (aircrafts). Первая таблица-копия будет предназначена для хранения данных, взятых из таблицы-прототипа, а вторая таблица-копия будет использоваться в качестве журнальной таблицы: будем записывать в нее все операции, проведенные с первой таблицей.

Создадим первую таблицу, причем, копировать данные из постоянной таблицы aircrafts не будем, о чем говорит предложение WITH NO DATA. Если бы мы решили скопировать в новую таблицу и все строки, содержащиеся в таблице-прототипе, тогда в команде CREATE TABLE мы могли бы использовать предложение WITH DATA или вообще не указывать его: по умолчанию строки копируются в создаваемую таблицу.

```
CREATE TEMP TABLE aircrafts_tmp AS
  SELECT * FROM aircrafts WITH NO DATA;
```

Наложим на таблицу необходимые ограничения: они не создаются при копировании таблицы. При массовом вводе данных гораздо более эффективным с точки зрения производительности было бы сначала добавить строки в таблицу, а уже потом накладывать ограничения на нее. Однако в нашем случае речь о массовом вводе не идет, поэтому мы начнем с наложения ограничений, а уже потом добавим строки с таблицу.

```
ALTER TABLE aircrafts_tmp
  ADD PRIMARY KEY ( aircraft_code );

ALTER TABLE aircrafts_tmp
  ADD UNIQUE ( model );
```

Теперь создадим вторую таблицу, копировать данные из постоянной таблицы aircrafts в нее также не будем.

```
CREATE TEMP TABLE aircrafts_log AS
  SELECT * FROM aircrafts WITH NO DATA;
```

Ограничения в виде первичного и уникального ключей этой таблице не требуются, но потребуются еще два столбца: первый будет содержать дату/время выполнения операции над таблицей aircrafts_tmp, а второй — наименование этой операции (INSERT, UPDATE или DELETE).

```
ALTER TABLE aircrafts_log
  ADD COLUMN when_add timestamp;

ALTER TABLE aircrafts_log
  ADD COLUMN operation text;
```

Поскольку в рассматриваемой ситуации копировать данные из постоянных таблиц во временные не требуется, то в качестве альтернативного способа создания временных таблиц можно было бы воспользоваться командой CREATE TEMP TABLE с предложением LIKE. Например:

```
CREATE TEMP TABLE aircrafts_tmp
( LIKE aircrafts INCLUDING CONSTRAINTS INCLUDING INDEXES );
```

Но так как уникального индекса по столбцу model в таблице aircrafts нет, то для временной таблицы его пришлось бы сформировать с помощью команды ALTER TABLE, как и при использовании первого способа ее создания. Добавим, что предложение LIKE можно применять для создания не только временных таблиц, но и постоянных.

Поскольку у нас есть журнальная таблица aircrafts_log, мы можем все операции с таблицей aircrafts_tmp записывать в таблицу aircrafts_log, т. е. вести историю изменений данных в таблице aircrafts_tmp.

Начнем работу с того, что скопируем в таблицу aircrafts_tmp все данные из таблицы aircrafts. Для выполнения не только «полезной» работы, но и ведения журнала изменений воспользуемся **командой INSERT с общим табличным выражением**.

Вообще, при классическом подходе для ведения учета изменений, внесенных в таблицы, используют триггеры или правила (rules), но их рассмотрение выходит за рамки настоящего пособия. Поэтому наш пример нужно рассматривать как иллюстрацию возможностей общих табличных выражений (CTE), а не как единственно верный подход.

```
WITH add_row AS
( INSERT INTO aircrafts_tmp
  SELECT * FROM aircrafts
  RETURNING *
)
INSERT INTO aircrafts_log
  SELECT add_row.aircraft_code, add_row.model, add_row.range,
     CURRENT_TIMESTAMP, 'INSERT'
  FROM add_row;
```

```
INSERT 0 9
```

Давайте рассмотрим эту команду более подробно. Обратите внимание, что вся «полезная» работа выполняется в рамках конструкции WITH add_row AS (...). Здесь строки

с помощью команды SELECT выбираются из таблицы aircrafts и вставляются в таблицу aircrafts_tmp. При вставке строк, выбранных из одной таблицы, в другую таблицу необходимо, чтобы число атрибутов и их типы данных во вставляемых строках были согласованы с числом столбцов и их типами данных в целевой таблице. Завершается конструкция WITH add_row AS (...) предложением RETURNING *, которое просто возвращает внешнему запросу все строки, успешно добавленные в таблицу aircrafts_tmp. Конечно же, при этом из таблицы aircrafts_tmp добавленные строки никуда не исчезают. Запрос получает имя add_row, на которое может ссылаться внешний запрос, когда он «хочет» обратиться к строкам, возвращенным с помощью предложения RETURNING *.

Теперь обратимся к внешнему запросу. В нем также присутствует команда INSERT, которая получает данные для ввода в таблицу aircrafts_log от запроса SELECT. А этот запрос, в свою очередь, получает данные от временной таблицы add_row, указанной в предложении FROM. Поскольку в предложении RETURNING мы указали в качестве возвращаемого значения «*», то будут возвращены все столбцы таблицы aircrafts_tmp, т. е. той таблицы, в которую строки были *вставлены*. Следовательно, в команде SELECT внешнего запроса можно ссылаться на имена этих столбцов:

```
SELECT add_row.aircraft_code, add_row.model, add_row.range, ...
```

Поскольку в таблице aircrafts_log существует еще два столбца, то для них мы дополнительно передаем значения CURRENT_TIMESTAMP и 'INSERT'.

Проверим, что получилось:

```
SELECT * FROM aircrafts_tmp ORDER BY model;
```

```
 aircraft_code |        model        | range
---------------+---------------------+-------
 319           | Airbus A319-100     |  6700
 320           | Airbus A320-200     |  5700
 321           | Airbus A321-200     |  5600
 733           | Boeing 737-300      |  4200
 763           | Boeing 767-300      |  7900
 773           | Boeing 777-300      | 11100
 CR2           | Bombardier CRJ-200  |  2700
 CN1           | Cessna 208 Caravan  |  1200
 SU9           | Sukhoi SuperJet-100 |  3000
(9 строк)
```

Проверим также и содержимое журнальной таблицы:

```
SELECT * FROM aircrafts_log ORDER BY model;
```

```
--[ RECORD 1 ]-+--------------------------
 aircraft_code | 319
 model         | Airbus A319-100
 range         | 6700
 when_add      | 2017-01-31 18:28:49.230179
 operation     | INSERT
--[ RECORD 2 ]-+--------------------------
 aircraft_code | 320
 model         | Airbus A320-200
 range         | 5700
```

```
 when_add      | 2017-01-31 18:28:49.230179
 operation     | INSERT
...
```

При вставке новых строк могут возникать ситуации, когда нарушается ограничение первичного или уникального ключей, поскольку вставляемые строки могут иметь значения ключевых атрибутов, совпадающие с теми, что уже имеются в таблице. Для таких случаев предусмотрено специальное средство — предложение ON CONFLICT, оно предусматривает два варианта действий на выбор программиста. Первый вариант — отменять добавление новой строки, для которой имеет место конфликт значений ключевых атрибутов, и при этом не порождать сообщения об ошибке. Второй вариант заключается в замене операции добавления новой строки операцией обновления существующей строки, с которой конфликтует добавляемая строка.

Начнем с первого варианта. Попробуем добавить строку, которая гарантированно будет конфликтовать с уже существующей строкой, причем, как по первичному ключу aircraft_code, так и по уникальному ключу model.

```
WITH add_row AS
  ( INSERT INTO aircrafts_tmp
      VALUES ( 'SU9', 'Sukhoi SuperJet-100', 3000 )
      ON CONFLICT DO NOTHING
      RETURNING *
  )
INSERT INTO aircrafts_log
  SELECT add_row.aircraft_code, add_row.model, add_row.range,
         CURRENT_TIMESTAMP, 'INSERT'
    FROM add_row;
```

Обратите внимание, что не будет выведено никаких сообщений об ошибках, как это и предполагалось. Строка добавлена не будет:

```
INSERT 0 0
```

Нужно учитывать, что это сообщение относится к таблице aircrafts_log, т. е. к команде в главном запросе, а не в общем табличном выражении WITH add_row AS (...), в котором мы работаем с таблицей aircrafts_tmp. Проверьте, не была ли добавлена строка в таблицу aircrafts_tmp.

В том случае, когда в предложении ON CONFLICT не указана дополнительная информация об именах столбцов или ограничений, по которым предполагается возможный конфликт, проверка выполняется по первичному ключу и по уникальным ключам.

Давайте укажем конкретный столбец для проверки конфликтующих значений. Пусть это будет столбец aircraft_code, т. е. первичный ключ. Для упрощения команды не будем использовать общее табличное выражение. Добавляемая строка будет иметь конфликт с существующей строкой как по столбцу aircraft_code, так и по столбцу model.

```
INSERT INTO aircrafts_tmp
  VALUES ( 'SU9', 'Sukhoi SuperJet-100', 3000 )
  ON CONFLICT ( aircraft_code ) DO NOTHING
  RETURNING *;
```

Получим только такое сообщение:

```
 aircraft_code | model | range
---------------+-------+-------
(0 строк)

INSERT 0 0
```

Это сообщение было выведено потому, что в команду включено предложение RETURNING *. Сообщение о дублировании значений столбца model не выводится.

Давайте в команде INSERT изменим значение столбца aircraft_code, чтобы оно стало уникальным:

```
INSERT INTO aircrafts_tmp
  VALUES ( 'S99', 'Sukhoi SuperJet-100', 3000 )
  ON CONFLICT ( aircraft_code ) DO NOTHING
  RETURNING *;
```

Поскольку конфликта по столбцу aircraft_code нет, то далее проверяется выполнение требования уникальности по столбцу model. В результате мы получим традиционное сообщение об ошибке, относящееся к столбцу model:

```
ОШИБКА: повторяющееся значение ключа нарушает ограничение уникальности
  ↪  "aircrafts_tmp_model_key"
ПОДРОБНОСТИ: Ключ "(model)=(Sukhoi SuperJet-100)" уже существует.
```

Теперь рассмотрим второй вариант обработки предложения ON CONFLICT, когда операция вставки новой строки заменяется операцией обновления существующей строки, с которой и возник конфликт значений столбцов. Для реализации этой возможности служит предложение DO UPDATE.

Давайте модифицируем команду и добавим предложение DO UPDATE. Выберем такую политику для работы с таблицей aircrafts_tmp: если при вставке новой строки имеет место дублирование по атрибутам первичного ключа со строкой, находящейся в таблице, тогда мы будем обновлять значения всех остальных атрибутов в этой строке, независимо от того, совпадают ли они со значениями в новой строке или нет. В качестве примера сделаем так: в добавляемой строке значение атрибута model сделаем отличающимся от того, которое уже есть в таблице (вместо «Sukhoi SuperJet-100» будет «Sukhoi SuperJet»), а значение атрибута range оставим без изменений (3000).

Внесем в команду еще одно изменение, а именно: вместо имени столбца, образующего первичный ключ, с помощью предложения ON CONSTRAINT укажем наименование ограничения первичного ключа.

Вот так выглядит команда с предложением DO UPDATE:

```
INSERT INTO aircrafts_tmp
  VALUES ( 'SU9', 'Sukhoi SuperJet', 3000 )
  ON CONFLICT ON CONSTRAINT aircrafts_tmp_pkey
  DO UPDATE SET model = excluded.model,
                range = excluded.range
  RETURNING *;
```

Поскольку мы включили в команду предложение RETURNING *, то СУБД сообщит о том, какие значения получат атрибуты обновленной строки. Как и планировалось, изменилось только значение атрибута model.

```
 aircraft_code |      model      | range
---------------+-----------------+-------
 SU9           | Sukhoi SuperJet |  3000
(1 строка)
```

В случае конфликта по столбцу aircraft_code будет обновлена та строка в таблице aircrafts_tmp, с которой конфликтовала вновь добавляемая строка. В результате новая строка добавлена не будет, а будет обновлено значение столбца model в строке, уже находящейся в таблице. А где PostgreSQL возьмет значение для использования в команде UPDATE? Это значение будет взято из специальной таблицы excluded, которая поддерживается самой СУБД. В этой таблице хранятся все строки, предлагаемые для вставки в рамках текущей команды INSERT. Вот это значение — excluded.model. Значение столбца range также будет обновлено, но его новое значение — excluded.range — совпадает со старым.

Обратите внимание, что в предложении DO UPDATE не указывается имя таблицы, т. к. таблица будет та же самая, которая указана в предложении INSERT.

Предложение ON CONFLICT DO UPDATE гарантирует атомарное выполнение операции вставки или обновления строк. Атомарность означает, что проверка наличия конфликта и последующее обновление выполняются как неделимая операция, т. е. другие транзакции не могут изменить значение столбца, вызывающее конфликт, так, чтобы в результате конфликт исчез и уже стало возможным выполнить операцию INSERT, а не UPDATE, или, наоборот, в случае отсутствия конфликта он вдруг появился, и уже операция INSERT стала бы невозможной. Такая атомарная операция даже имеет название UPSERT — «UPDATE или INSERT».

Для массового ввода строк в таблицы используется команда COPY. Эта команда может копировать данные из файла в таблицу. Причем, в качестве файла может служить и стандартный ввод. Хотя в этом разделе пособия мы, в основном, говорим о вставке строк в таблицы, но нужно сказать и о том, что эта команда может также выводить данные из таблиц в файлы и на стандартный вывод.

В качестве примера ввода данных из файла давайте добавим две строки в таблицу aircrafts_tmp. Сначала необходимо подготовить текстовый файл, содержащий новые данные. В этом файле каждая строка соответствует одной строке таблицы. Значения атрибутов разделяются символами табуляции, поэтому пробелы, которые в нашем примере есть в столбце model, можно вводить в файл без каких-либо дополнительных экранирующих символов. Заключать строковые значения в одинарные кавычки не нужно, иначе они также будут введены в таблицу. Завершить файл нужно строкой, содержащей только символы «\.». Получим файл следующего содержания:

```
IL9    Ilyushin IL96        9800
I93    Ilyushin IL96-300    9800
\.
```

Теперь нужно ввести команду COPY, указав полный путь к вашему файлу:

```
COPY aircrafts_tmp FROM '/home/postgres/aircrafts.txt';
```

В результате будет выведено сообщение об успешном добавлении двух строк:

```
COPY 2
```

Давайте проверим, что получилось:

```
SELECT * FROM aircrafts_tmp;
```

Вы увидите, что новые строки были добавлены, но все те, что уже находились в таблице, удалены не были.

При добавлении строк с помощью команды COPY выполняются проверки всех ограничений, наложенных на таблицу, поэтому ввести дублирующие данные не получится.

Эту команду можно использовать и для вывода данных из таблицы в файл:

```
COPY aircrafts_tmp TO '/home/postgres/aircrafts_tmp.txt'
  WITH ( FORMAT csv );
```

Предложение FORMAT csv говорит о том, что при выводе данных значения столбцов разделяются запятыми (CSV — Comma Separated Values).

Получим файл такого вида:

```
773,Boeing 777-300,11100
763,Boeing 767-300,7900
SU9,Sukhoi SuperJet-100,3000
...
```

Если формат не указывать, то данные будут выведены с использованием символов табуляции в качестве разделителей значений атрибутов.

## 7.2 Обновление строк в таблицах

Команда UPDATE предназначена для обновления данных в таблицах. Начнем с того, что покажем, как и при изучении команды INSERT, как можно организовать запись выполненных операций в журнальную таблицу. Эта команда аналогична команде, уже рассмотренной в предыдущем разделе. В ней также «полезная» работа выполняется в общем табличном выражении, а запись в журнальную таблицу — в основном запросе.

```
WITH update_row AS
  ( UPDATE aircrafts_tmp
      SET range = range * 1.2 WHERE model ~ '^Bom'
      RETURNING *
  )
INSERT INTO aircrafts_log
  SELECT ur.aircraft_code, ur.model, ur.range,
         CURRENT_TIMESTAMP, 'UPDATE'
    FROM update_row ur;
```

Выполнив команду, в ответ получим сообщение

```
INSERT 0 1
```

Напомним, что выведенное сообщение относится непосредственно к внешнему запросу, в котором выполняется операция INSERT, добавляющая строку в журнальную таблицу. Конечно, если бы строка в таблице aircrafts_tmp не была успешно обновлена, тогда предложение RETURNING * не возвратило бы внешнему запросу ни одной строки, и, следовательно, тогда просто не было бы данных для формирования новой строки в таблице aircrafts_log.

При использовании команды UPDATE в общем табличном выражении нужно учитывать, что главный запрос может получить доступ к обновленным данным *только через временную таблицу*, которую формирует предложение RETURNING:

```
...
FROM update_row ur;
...
```

Можно выполнить выборку из журнальной таблицы aircrafts_log, чтобы посмотреть — правда, не очень длинную — историю изменений строки с описанием самолета Bombardier CRJ-200.

```
SELECT * FROM aircrafts_log
  WHERE model ~ '^Bom' ORDER BY when_add;
```

```
--[ RECORD 1 ]-+--------------------------
 aircraft_code | CR2
 model         | Bombardier CRJ-200
 range         | 2700
 when_add      | 2017-02-05 00:27:38.591958
 operation     | INSERT
--[ RECORD 2 ]-+--------------------------
 aircraft_code | CR2
 model         | Bombardier CRJ-200
 range         | 3240
 when_add      | 2017-02-05 00:27:56.688933
 operation     | UPDATE
```

Представим себе такую ситуацию: руководство компании хочет иметь возможность видеть динамику продаж билетов по всем направлениям, а именно: общее число проданных билетов и дату/время последнего увеличения их числа для конкретного направления.

Создадим временную таблицу, которую назовем tickets_directions. В ней будет четыре столбца:

- город отправления — departure_city;
- город прибытия — arrival_city;
- дата/время последнего увеличения числа проданных билетов — last_ticket_time;
- число проданных билетов на этот момент времени по данному направлению — tickets_num.

Создадим таблицу с помощью запроса к представлению «Маршруты» (routes) и заполним данными, однако в ней сначала будет только два первых столбца.

```
CREATE TEMP TABLE tickets_directions AS
  SELECT DISTINCT departure_city, arrival_city
    FROM routes;
```

Ключевое слово DISTINCT является здесь обязательным: ведь нам нужны только уникальные пары городов отправления и прибытия.

Добавим еще два столбца и заполним столбец-счетчик нулевыми значениями.

```
ALTER TABLE tickets_directions
  ADD COLUMN last_ticket_time timestamp;

ALTER TABLE tickets_directions
  ADD COLUMN tickets_num integer DEFAULT 0;
```

Поскольку PostgreSQL не требует обязательного создания первичного ключа, то не будем создавать его. Это не помешает нам однозначно идентифицировать строки в таблице tickets_directions.

Поскольку в команде UPDATE нет предложения WHERE, в котором было бы условие, ограничивающее множество обновляемых строк, то будут обновлены все строки таблицы — во все будет записано значение 0 в столбец tickets_num.

Для того чтобы не усложнять изложение материала, создадим временную таблицу, являющуюся аналогом таблицы «Перелеты» (ticket_flights), однако без внешних ключей. Поэтому мы сможем добавлять в нее строки, не заботясь о добавлении строк в таблицы «Билеты» (tickets) и «Бронирования» (bookings). Тем не менее, первичный ключ все же создадим, чтобы продемонстрировать, что в случае попытки ввода строк с дубликатными значениями первичного ключа значения счетчиков в таблице tickets_directions наращиваться не будут.

```
CREATE TEMP TABLE ticket_flights_tmp AS
  SELECT * FROM ticket_flights WITH NO DATA;

ALTER TABLE ticket_flights_tmp
  ADD PRIMARY KEY ( ticket_no, flight_id );
```

Теперь представим команду, которая и будет добавлять новую запись о продаже билета и увеличивать значение счетчика проданных билетов в таблице tickets_directions.

```
WITH sell_ticket AS
  ( INSERT INTO ticket_flights_tmp
      ( ticket_no, flight_id, fare_conditions, amount )
      VALUES ( '1234567890123', 30829, 'Economy', 12800 )
      RETURNING *
  )
  UPDATE tickets_directions td
    SET last_ticket_time = CURRENT_TIMESTAMP,
        tickets_num = tickets_num + 1
    WHERE ( td.departure_city, td.arrival_city ) =
       ( SELECT departure_city, arrival_city
           FROM flights_v
           WHERE flight_id = ( SELECT flight_id FROM sell_ticket )
    );
```

```
UPDATE 1
```

Этот запрос работает следующим образом. Добавление новой записи о бронировании авиаперелета производится в общем табличном выражении, а наращивание соответствующего счетчика — в главном запросе. Поскольку в общем табличном выражении присутствует предложение RETURNING *, значения атрибутов добавленной строки будут доступны в главном запросе посредством обращения к временной таблице sell_ticket. Конечно, если строка фактически не будет добавлена из-за дублирования значения первичного ключа, тогда будет сгенерировано сообщение об ошибке, в результате главный запрос выполнен не будет, следовательно, таблица tickets_directions не будет обновлена.

В главном запросе мы обновляем всего два атрибута, причем, значение атрибута tickets_num может увеличиться только на единицу, поскольку в таблицу ticket_flights_tmp добавляется одна строка. Остается выяснить, каким образом мы сможем определить ту строку в таблице tickets_directions, атрибуты которой нужно обновить. Нам требуется на основе значения идентификатора рейса flight_id, на который был забронирован билет (перелет), определить города отправления и прибытия, которые как раз и идентифицируют строку в таблице tickets_directions. Эти три атрибута присутствуют в представлении flights_v. Подзапрос обращается к этому представлению, а вложенный подзапрос возвращает значение идентификатора рейса flight_id, на который был забронирован билет (перелет). Назначение вложенного подзапроса в том, чтобы в условии WHERE flight_id = ... не дублировать значение атрибута flight_id, использованное в команде INSERT (в данном примере это 30829). Тем самым должен быть снижен риск ошибки при вводе данных.

Обратите внимание, что подзапрос в предложении WHERE возвращает два столбца, и сравнение выполняется также сразу с двумя столбцами.

Посмотрим, что получилось:

```
SELECT * FROM tickets_directions WHERE tickets_num > 0;
```

```
--[ RECORD 1 ]----+---------------------------
 departure_city   | Сочи
 arrival_city     | Красноярск
 last_ticket_time | 2017-02-04 21:15:32.903687
 tickets_num      | 1
```

Представим другой вариант этой команды. Его принципиальное отличие от первого варианта заключается в том, что для определения обновляемой строки в таблице tickets_directions используется **операция соединения таблиц**. Здесь в главном запросе UPDATE присутствует предложение FROM, однако в этом предложении указывается только таблица (представление) flights_v, а таблицу tickets_directions в предложение FROM включать не нужно, хотя она и участвует в выполнении соединения таблиц. Конечно, в предложении SET присваивать новые значения можно только атрибутам таблицы tickets_directions, поскольку именно она приведена в предложении UPDATE.

```
WITH sell_ticket AS
  ( INSERT INTO ticket_flights_tmp
      (ticket_no, flight_id, fare_conditions, amount )
      VALUES ( '1234567890123', 7757, 'Economy', 3400 )
      RETURNING *
```

```
  )
UPDATE tickets_directions td
  SET last_ticket_time = CURRENT_TIMESTAMP,
      tickets_num = tickets_num + 1
  FROM flights_v f
  WHERE td.departure_city = f.departure_city
    AND td.arrival_city   = f.arrival_city
    AND f.flight_id = ( SELECT flight_id FROM sell_ticket );
```

```
UPDATE 1
```

Посмотрим, что получилось.

```
SELECT * FROM tickets_directions WHERE tickets_num > 0;
```

```
--[ RECORD 1 ]----+---------------------------
 departure_city   | Сочи
 arrival_city     | Красноярск
 last_ticket_time | 2017-02-04 21:15:32.903687
 tickets_num      | 1
--[ RECORD 2 ]----+---------------------------
 departure_city   | Москва
 arrival_city     | Сочи
 last_ticket_time | 2017-02-04 21:18:40.353408
 tickets_num      | 1
```

Чтобы увидеть комбинированную строку, которая получилась при соединении таблиц tickets_directions и flights_v, можно включить в команду UPDATE предложение RETURNING *.

## 7.3 Удаление строк из таблиц

Начнем рассмотрение команды DELETE, предназначенной для удаления данных из таблиц, с того, что, как и при изучении команды INSERT, покажем, как можно организовать запись выполненных операций в журнальную таблицу. Эта команда аналогична команде, уже рассмотренной в предыдущем разделе. В ней также «полезная» работа выполняется в общем табличном выражении, а запись в журнальную таблицу — в основном запросе.

```
WITH delete_row AS
  ( DELETE FROM aircrafts_tmp
      WHERE model ~ '^Bom'
      RETURNING *
  )
INSERT INTO aircrafts_log
  SELECT dr.aircraft_code, dr.model, dr.range,
         CURRENT_TIMESTAMP, 'DELETE'
    FROM delete_row dr;
```

Выполнив команду, в ответ получим сообщение

```
INSERT 0 1
```

Напомним, что выведенное сообщение относится непосредственно к внешнему запросу, в котором выполняется операция INSERT, добавляющая строку в журнальную таблицу.

Посмотрим историю изменений строки с описанием самолета Bombardier CRJ-200:

```
SELECT * FROM aircrafts_log
  WHERE model ~ '^Bom' ORDER BY when_add;

--[ RECORD 1 ]-+--------------------------
 aircraft_code | CR2
 model         | Bombardier CRJ-200
 range         | 2700
 when_add      | 2017-02-05 00:27:38.591958
 operation     | INSERT
--[ RECORD 2 ]-+--------------------------
 aircraft_code | CR2
 model         | Bombardier CRJ-200
 range         | 3240
 when_add      | 2017-02-05 00:27:56.688933
 operation     | UPDATE
--[ RECORD 3 ]-+--------------------------
 aircraft_code | CR2
 model         | Bombardier CRJ-200
 range         | 3240
 when_add      | 2017-02-05 00:34:59.510911
 operation     | DELETE
```

Для удаления конкретных строк из данной таблицы можно использовать информацию не только из нее, но также и из других таблиц. Выбирать строки для удаления можно двумя способами: использовать подзапросы к этим таблицам в предложении WHERE или указать дополнительные таблицы в предложении USING, а затем в предложении WHERE записать условия соединения таблиц. Поскольку первый способ является традиционным, то мы покажем второй из них.

Предположим, что руководство авиакомпании решило удалить из парка самолетов машины компаний Boeing и Airbus, имеющие наименьшую дальность полета.

Решим эту задачу следующим образом. В общем табличном выражении с помощью условия model ~'^Airbus' OR model ~'^Boeing' в предложении WHERE отберем модели только компаний Boeing и Airbus. Затем воспользуемся оконной функцией rank и произведем ранжирование моделей каждой компании по возрастанию дальности полета. Те модели, ранг которых окажется равным 1, будут иметь наименьшую дальность полета.

В предложении USING сформируем соединение таблицы aircrafts_tmp с временной таблицей min_ranges, а затем в предложении WHERE зададим условия для отбора строк.

```
WITH min_ranges AS
  ( SELECT aircraft_code,
           rank() OVER (
             PARTITION BY left( model, 6 )
             ORDER BY range
           ) AS rank
```

```
      FROM aircrafts_tmp
      WHERE model ~ '^Airbus' OR model ~ '^Boeing'
  )
DELETE FROM aircrafts_tmp a
  USING min_ranges mr
  WHERE a.aircraft_code = mr.aircraft_code
    AND mr.rank = 1
  RETURNING *;
```

Мы включили в команду DELETE предложение RETURNING *, чтобы показать, как выглядят комбинированные строки, сформированные с помощью предложения USING. Конечно, удаляются не комбинированные строки, а только оригинальные строки из таблицы aircrafts_tmp.

```
 aircraft_code |       model      | range | aircraft_code | rank
---------------+------------------+-------+---------------+------
 321           | Airbus A321-200 |  5600 | 321           |    1
 733           | Boeing 737-300  |  4200 | 733           |    1
(2 строки)
```

В заключение этого раздела упомянем еще команду TRUNCATE, которая позволяет быстро удалить все строки из таблицы. Следующие две команды позволяют удалить все строки из таблицы aircrafts_tmp:

```
DELETE FROM aircrafts_tmp;
TRUNCATE aircrafts_tmp;
```

Однако команда TRUNCATE работает быстрее.

## Контрольные вопросы и задания

1. Добавьте в определение таблицы aircrafts_log значение по умолчанию CURRENT_TIMESTAMP и соответствующим образом измените команды INSERT, приведенные в тексте главы.
2. В предложении RETURNING можно указывать не только символ «*», означающий выбор всех столбцов таблицы, но также и более сложные выражения, сформированные на основе этих строк. В тексте главы мы копировали содержимое таблицы «Самолеты» (aircrafts) в таблицу aircrafts_tmp. В том запросе в предложении RETURNING мы использовали именно «*». Однако возможен и другой вариант запроса:

   ```
   WITH add_row AS
     ( INSERT INTO aircrafts_tmp
         SELECT * FROM aircrafts
         RETURNING aircraft_code, model, range,
                   CURRENT_TIMESTAMP, 'INSERT'
     )
   INSERT INTO aircrafts_log
     SELECT ? FROM add_row;
   ```

   Что нужно написать в этом запросе вместо вопросительного знака?

3. Если бы мы для копирования данных из таблицы «Самолеты» (aircrafts) в таблицу aircrafts_tmp использовали команду INSERT без общего табличного выражения

   ```
   INSERT INTO aircrafts_tmp SELECT * FROM aircrafts;
   ```

   то в качестве выходного результата мы увидели бы сообщение

   ```
   INSERT 0 9
   ```

   Как вы думаете, что будет выведено, если дополнить команду предложением RETURNING * ?

   ```
   INSERT INTO aircrafts_tmp SELECT * FROM aircrafts RETURNING *;
   ```

   Проверьте ваши предположения на практике. Подумайте, каким образом можно использовать выведенный результат?

4. В тексте главы в предложениях ON CONFLICT команды INSERT мы использовали только выражения, состоящие из имени одного столбца. Однако в таблице «Места» (seats) первичный ключ является составным и включает два столбца. Напишите команду INSERT для вставки новой строки в эту таблицу и предусмотрите возможный конфликт добавляемой строки со строкой, уже имеющейся в таблице. Сделайте два варианта предложения ON CONFLICT: первый — с использованием перечисления имен столбцов для проверки наличия дублирования, второй — с использованием предложения ON CONSTRAINT.

   Для того чтобы не изменить содержимое таблицы «Места», создайте ее копию и выполняйте все эти эксперименты с таблицей-копией.

5. В предложении DO UPDATE команды INSERT может использоваться условие WHERE. Самостоятельно ознакомьтесь с этой возможностью с помощью документации и напишите такую команду INSERT.

6. Команда COPY по умолчанию ожидает получения вводимых данных в формате text, когда значения данных разделяются символами табуляции. Однако можно представлять входные данные в формате CSV (Comma Separated Values), т. е. использовать в качестве разделителя запятую.

   ```
   COPY aircrafts_tmp FROM STDIN WITH ( FORMAT csv );
   ```

   ```
   Вводите данные для копирования, разделяя строки переводом строки.
   Закончите ввод строкой '\.'.
   >> IL9, Ilyushin IL96, 9800
   >> I93, Ilyushin IL96-300, 9800
   >> \.
   ```

   ```
   SELECT * FROM aircrafts_tmp;
   ```

   ```
    aircraft_code |        model         | range
   ---------------+----------------------+-------
   ...
    CN1           |  Cessna 208 Caravan |  1200
    CR2           |  Bombardier CRJ-200 |  2700
    IL9           |  Ilyushin IL96      |  9800
    I93           |  Ilyushin IL96-300  |  9800
   (11 строк)
   ```

Как вы думаете, почему при выводе данных из таблицы вновь введенные значения в столбце model оказались смещены вправо?

7. Команда COPY позволяет получить входные данные из файла и поместить их в таблицу. Этот файл должен быть доступен тому пользователю операционной системы, от имени которого запущен серверный процесс, как правило, это пользователь postgres.

   Подготовьте файл, например, /home/postgres/aircrafts_tmp.csv, имеющий такую структуру:

   – каждая строка файла соответствует одной строке таблицы aircrafts_tmp;

   – значения данных в строке файла разделяются запятыми.

   Например:

   ```
   773,Boeing 777-300,11100
   763,Boeing 767-300,7900
   SU9,Sukhoi SuperJet-100,3000
   ```

   Введите в этот файл данные о нескольких самолетах, причем часть из них уже должна быть представлена в таблице, а часть — нет.

   Поскольку при выполнении команды COPY выполняются проверки всех ограничений целостности, наложенных на таблицу, то дублирующие строки добавлены, конечно же, не будут. А как вы думаете, строки, содержащиеся в этом же файле, но отсутствующие в таблице, будут добавлены или нет? Проверьте свою гипотезу, выполнив команду для вставки строк в таблицу из этого файла:

   ```
   COPY aircrafts_tmp
     FROM '/home/postgres/aircrafts_tmp.csv'
     WITH ( FORMAT csv );
   ```

8.* В тексте главы был приведен запрос, предназначенный для учета числа билетов, проданных по всем направлениям на текущую дату. Однако тот запрос был рассчитан на одновременное добавление только одной записи в таблицу «Перелеты» (ticket_flights_tmp). Ниже мы предложим более универсальный запрос, который предусматривает возможность единовременного ввода нескольких записей о перелетах, выполняемых на различных рейсах.

   Для проверки работоспособности предлагаемого запроса выберем несколько рейсов по маршрутам: Красноярск — Москва, Москва — Сочи, Сочи — Москва, Сочи — Красноярск. Для определения идентификаторов рейсов сформируем вспомогательный запрос, в котором даты начала и конца рассматриваемого периода времени зададим с помощью функции bookings.now. Использование этой функции необходимо, поскольку в будущих версиях базы данных могут быть представлены другие диапазоны дат.

   ```
   SELECT flight_no, flight_id, departure_city, arrival_city,
          scheduled_departure
     FROM flights_v
     WHERE scheduled_departure
         BETWEEN bookings.now() AND bookings.now() + INTERVAL '15 days'
       AND ( departure_city, arrival_city ) IN
         ( ( 'Красноярск', 'Москва' ), ( 'Москва', 'Сочи'),
   ```

```
      ( 'Сочи', 'Москва' ), ( 'Сочи', 'Красноярск' )
    )
  ORDER BY departure_city, arrival_city, scheduled_departure;
```

Обратите внимание на предикат IN: в нем используются не индивидуальные значения, а пары значений.

Предположим, что в течение указанного интервала времени пассажир планирует совершить перелеты по маршруту: Красноярск — Москва, Москва — Сочи, Сочи — Москва, Москва — Сочи, Сочи — Красноярск. Выполнив вспомогательный запрос, выберем следующие идентификаторы рейсов (в этом же порядке): 13829, 4728, 30523, 7757, 30829.

```
WITH sell_tickets AS
  ( INSERT INTO ticket_flights_tmp
      ( ticket_no, flight_id, fare_conditions, amount )
      VALUES ( '1234567890123', 13829, 'Economy', 10500 ),
             ( '1234567890123',  4728, 'Economy',  3400 ),
             ( '1234567890123', 30523, 'Economy',  3400 ),
             ( '1234567890123',  7757, 'Economy',  3400 ),
             ( '1234567890123', 30829, 'Economy', 12800 )
      RETURNING *
  )
UPDATE tickets_directions td
  SET last_ticket_time = CURRENT_TIMESTAMP,
      tickets_num = tickets_num +
         ( SELECT count( * )
             FROM sell_tickets st, flights_v f
             WHERE st.flight_id = f.flight_id
               AND f.departure_city = td.departure_city
               AND f.arrival_city   = td.arrival_city
         )
  WHERE ( td.departure_city, td.arrival_city ) IN (
              SELECT departure_city, arrival_city
                FROM flights_v
                WHERE flight_id IN (
                  SELECT flight_id
                    FROM sell_tickets
                )
        );
```

```
UPDATE 4
```

В этой версии запроса предусмотрен единовременный ввод нескольких строк в таблицу ticket_flights_tmp, причем, перелеты могут выполняться на различных рейсах. Поэтому необходимо каким-то образом преобразовать список идентификаторов этих рейсов в множество пар «город отправления — город прибытия», поскольку именно для таких пар и ведется подсчет числа забронированных перелетов. Эта задача решается в предложении WHERE. Здесь вложенный подзапрос формирует список идентификаторов рейсов, а внешний подзапрос преобразует этот список в множество пар «город отправления — город прибытия». Затем с помощью предиката IN производится отбор строк таблицы tickets_directions для обновления.

Теперь обратимся к предложению SET. Подзапрос с функцией count вычисляет количество перелетов по *каждому* направлению. Это коррелированный подзапрос: он выполняется для каждой строки, отобранной в предложении WHERE. В нем используется соединение временной таблицы sell_tickets с представлением flights_v. Это нужно для того, чтобы подсчитать все перелеты, соответствующие паре атрибутов «город отправления — город прибытия», взятых из текущей обновляемой строки таблицы tickets_directions. Этот подзапрос позволяет учесть такой факт: рейсы могут иметь различные идентификаторы flight_id, но при этом соответствовать одному и тому же направлению, а в таблице tickets_directions учитываются именно направления.

В случае попытки повторного бронирования одного и того же перелета для данного пассажира, т. е. ввода строки с дубликатом первичного ключа, такая строка будет отвергнута, и будет сгенерировано сообщение об ошибке. В таком случае и таблица tickets_directions не будет обновлена.

Давайте посмотрим, что изменилось в таблице tickets_directions.

```
SELECT departure_city AS dep_city, arrival_city AS arr_city,
       last_ticket_time, tickets_num AS num
  FROM tickets_directions
  WHERE tickets_num > 0
  ORDER BY departure_city, arrival_city;
```

По маршруту Москва — Сочи наш пассажир приобретал два билета, что и отражено в выборке.

```
  dep_city  |  arr_city  |      last_ticket_time      | num
------------+------------+----------------------------+-----
 Красноярск | Москва     | 2017-02-04 14:02:23.769443 |   1
 Москва     | Сочи       | 2017-02-04 14:02:23.769443 |   2
 Сочи       | Красноярск | 2017-02-04 14:02:23.769443 |   1
 Сочи       | Москва     | 2017-02-04 14:02:23.769443 |   1
(4 строки)
```

А это информация о каждом перелете, забронированном нашим пассажиром:

```
SELECT * FROM ticket_flights_tmp;
```

```
   ticket_no   | flight_id | fare_conditions |  amount
---------------+-----------+-----------------+----------
 1234567890123 |     13829 | Economy         | 10500.00
 1234567890123 |      4728 | Economy         |  3400.00
 1234567890123 |     30523 | Economy         |  3400.00
 1234567890123 |      7757 | Economy         |  3400.00
 1234567890123 |     30829 | Economy         | 12800.00
(5 строк)
```

Задание: модифицируйте запрос и таблицу tickets_directions так, чтобы учет числа забронированных перелетов по различным маршрутам выполнялся для каждого класса обслуживания: Economy, Business и Comfort.

9.* Предположим, что руководство нашей авиакомпании решило отказаться от использования самолетов компаний Boeing и Airbus, имеющих наименьшее количество пассажирских мест в салонах. Мы должны соответствующим образом откорректировать таблицу «Самолеты» (aircrafts_tmp).

Мы предлагаем такой алгоритм.

Шаг 1. Для каждой модели вычислить общее число мест в салоне.

Шаг 2. Используя оконную функцию rank, присвоить моделям ранги на основе числа мест (упорядочив их по возрастанию числа мест). Ранжирование выполняется *в пределах каждой компании-производителя*, т. е. для Boeing и для Airbus — отдельно. Ранг, равный 1, соответствует наименьшему числу мест.

Шаг 3. Выполнить удаление тех строк из таблицы aircrafts_tmp, которые удовлетворяют следующим требованиям: модель — Boeing или Airbus, а число мест в салоне — минимальное из всех моделей данной компании-производителя, т. е. модель имеет ранг, равный 1.

```
WITH aicrafts_seats AS
  ( SELECT aircraft_code, model, seats_num,
           rank() OVER (
             PARTITION BY left( model, strpos( model, ' ' ) - 1 )
             ORDER BY seats_num
           )
      FROM (
        SELECT a.aircraft_code, a.model, count( * ) AS seats_num
          FROM aircrafts_tmp a, seats s
          WHERE a.aircraft_code = s.aircraft_code
          GROUP BY 1, 2
      ) AS seats_numbers
)
DELETE FROM aircrafts_tmp a
  USING aicrafts_seats a_s
  WHERE a.aircraft_code = a_s.aircraft_code
    AND left( a.model, strpos( a.model, ' ' ) - 1 )
      IN ( 'Boeing', 'Airbus' )
    AND a_s.rank = 1
  RETURNING *;
```

Шаг 1 выполняется в подзапросе в предложении WITH. Шаг 2 выполняется в главном запросе в предложении WITH. Шаг 3 реализуется командой DELETE.

Обратите внимание, что название компании-производителя мы определяем путем взятия подстроки от значения атрибута model: от начала строки до пробельного символа (используем функции left и strpos).

Мы включили предложение RETURNING * для того, чтобы увидеть, какие именно модели были удалены.

Предложение WITH выдает такой результат:

```
 aircraft_code |        model         | seats_num | rank
---------------+----------------------+-----------+------
 319           | Airbus A319-100      |       116 |    1
 320           | Airbus A320-200      |       140 |    2
```

```
 321           | Airbus A321-200     |       170 |    3
 733           | Boeing 737-300      |       130 |    1
 763           | Boeing 767-300      |       222 |    2
 773           | Boeing 777-300      |       402 |    3
 CR2           | Bombardier CRJ-200  |        50 |    1
 CN1           | Cessna 208 Caravan  |        12 |    1
 SU9           | Sukhoi SuperJet-100 |        97 |    1
(9 строк)
```

Очевидно, что должны быть удалены модели с кодами 319 и 733.

После выполнения запроса получим такое сообщение (это работает предложение RETURNING *):

```
--[ RECORD 1 ]-+---------------
 aircraft_code | 319
 model         | Airbus A319-100
 range         | 6700
 aircraft_code | 319
 model         | Airbus A319-100
 seats_num     | 116
 rank          | 1
--[ RECORD 2 ]-+---------------
 aircraft_code | 733
 model         | Boeing 737-300
 range         | 4200
 aircraft_code | 733
 model         | Boeing 737-300
 seats_num     | 130
 rank          | 1

DELETE 2
```

Обратите внимание, что были выведены комбинированные строки, полученные при соединении таблицы aircrafts_tmp с временной таблицей aicrafts_seats, указанной в предложении USING. Но удалены были, конечно, строки из таблицы aircrafts_tmp.

Задание: предложите другой вариант решения этой задачи. Например, можно поступить так: оставить предложение WITH без изменений, из команды DELETE убрать предложение USING, а в предложении WHERE вместо соединения таблиц использовать подзапрос с предикатом IN для получения списка кодов удаляемых моделей самолетов.

Еще один вариант решения задачи связан с использованием представлений, которые мы рассматривали в главе 5. В данном случае можно создать представление на основе таблиц «Самолеты» (aircrafts) и «Места» (seats) и перенести конструкцию с функциями left и strpos в представление. В нем будут вычисляемые столбцы: company — «Компания — производитель самолетов» и seats_num — «Число мест».

```
CREATE VIEW aircrafts_seats AS
  ( SELECT a.aircraft_code, a.model,
        left( a.model, strpos( a.model, ' ' ) - 1 ) AS company,
        count( * ) AS seats_num
```

```
    FROM aircrafts a, seats s
    WHERE a.aircraft_code = s.aircraft_code
    GROUP BY 1, 2, 3
);
```

Имея это представление, можно использовать его в конструкции WITH. При этом вызов функции rank может упроститься:

```
rank() OVER ( PARTITION BY company ORDER BY seats_num )
```

Для выбора удаляемых строк в команде DELETE можно использовать, например, подзапрос в предикате IN. При этом не забывайте, что значение столбца rank для них будет равно 1.

Еще одна идея: для выбора минимальных значений числа мест в самолетах можно попытаться в качестве замены оконной функции rank использовать предложения LIMIT 1 и ORDER BY. В таком случае не потребуется также и функция min.

10.* В реальной работе иногда возникают ситуации, когда требуется быстро заполнить таблицу тестовыми данными. В таком случае можно воспользоваться командой INSERT с подзапросом. Конечно, число атрибутов и их типы данных в подзапросе SELECT должны быть такими, какие ожидает получить команда INSERT.

Продемонстрируем такой прием на примере таблицы «Места» (seats). Для того чтобы выполнить команду, приведенную в этом упражнении, нужно либо сначала удалить все строки из таблицы seats, чтобы можно было добавлять строки в эту таблицу

```
DELETE FROM seats;
```

либо создать копию этой таблицы

```
CREATE TABLE seats_tmp AS SELECT * FROM seats;
```

чтобы работать с копией.

Итак, как сформировать тестовые данные автоматическим способом? Для этого сначала нужно подготовить исходные данные, на основе которых и будут формироваться результирующие значения для вставки в таблицу «Места». В рамках реляционной модели наиболее естественным будет представление исходных данных в виде таблиц. Для формирования каждой строки таблицы «Места» нужно задать код модели самолета, класс обслуживания и номер места, который состоит из двух компонентов: номера ряда и буквенного идентификатора позиции в ряду. Поскольку размеры и компоновки салонов различаются, необходимо для каждой модели указать предельное число рядов кресел в салонах бизнес-класса и экономического класса, а также число кресел в каждом ряду. Это число можно задать с помощью указания буквенного идентификатора для самого последнего кресла в ряду. Например, если в ряду всего шесть кресел, тогда их буквенные обозначения будут такими: A, B, C, D, E, F. Таким образом, последней будет буква F. В салоне бизнес-класса число мест в ряду меньше, чем в салоне экономического класса, но для упрощения задачи примем эти числа одинаковыми. В результате получим первую исходную таблицу с атрибутами:

– код модели самолета;

– номер последнего ряда кресел в салоне бизнес-класса;

– номер последнего ряда кресел в салоне экономического класса;

– буква, обозначающая позицию последнего кресла в ряду.

Классы обслуживания также поместим в отдельную таблицу. В ней будет всего один атрибут — класс обслуживания.

Список номеров рядов также поместим в отдельную таблицу. В ней будет также всего один атрибут — номер ряда.

Также поступим и с буквенными обозначениями кресел в ряду. В этой таблице будет один атрибут — латинская буква, обозначающая позицию кресла.

В принципе можно было бы создать все четыре таблицы с помощью команды CREATE TABLE и ввести в них исходные данные, а затем использовать эти таблицы в команде SELECT. Но команда SELECT позволяет использовать в предложении FROM виртуальные таблицы, которые можно создавать с помощью предложения VALUES. Для этого непосредственно в текст команды записываются группы значений, представляющие собой строки такой виртуальной таблицы. Каждая такая строка заключается в круглые скобки. Вся эта конструкция получает имя таблицы и к ней прилагается список атрибутов. Это выглядит, например, так:

```
FROM (
  VALUES ( 'SU9', 3, 20, 'F' ),
         ( '773', 5, 30, 'I' ),
         ( '763', 4, 25, 'H' ),
         ( '733', 3, 20, 'F' ),
         ( '320', 5, 25, 'F' ),
         ( '321', 4, 20, 'F' ),
         ( '319', 3, 20, 'F' ),
         ( 'CN1', 0, 10, 'B' ),
         ( 'CR2', 2, 15, 'D' )
)
AS aircraft_info (
    aircraft_code, max_seat_row_business,
    max_seat_row_economy, max_letter
)
```

Здесь aircraft_info определяет имя виртуальной таблицы, а aircraft_code, max_seat_row_business, max_seat_row_economy, max_letter — имена ее атрибутов. Эти атрибуты можно использовать во всех частях команды SELECT, как если бы это были атрибуты обычной таблицы.

Остальные виртуальные таблицы создаются аналогичным способом.

Для соединения таблиц используется ключевое слово CROSS JOIN, хотя в данном случае вместо этого можно было просто поставить запятые.

Как это и бывает всегда, четыре таблицы образуют декартово произведение из своих строк, а затем на основе условия WHERE «лишние» строки отбрасываются. В этом условии используется условный оператор CASE. Он позволяет нам поставить допустимый номер ряда в зависимость от класса обслуживания:

```
WHERE
  CASE WHEN fare_condition = 'Business'
         THEN seat_row::integer <= max_seat_row_business
       WHEN fare_condition = 'Economy'
         THEN seat_row::integer > max_seat_row_business
          AND seat_row::integer <= max_seat_row_economy
```

В этом выражении используется операция приведения типа: seat_row::integer. Она необходима, т. к. в виртуальной таблице номера рядов представлены в виде символьных строк, а для выполнения сравнения числовых значений в данной ситуации нужен целый тип. При написании условного оператора нужно учесть, что в виртуальной таблице мы указали не количество рядов в бизнес-классе и экономическом классе, а номера *последних* рядов в этих классах. Поэтому возникает конструкция

```
THEN seat_row::integer > max_seat_row_business
 AND seat_row::integer <= max_seat_row_economy
```

Также проверяем еще одно условие, сравнивая символьные строки:

```
AND letter <= max_letter;
```

Последний этап в работе оператора SELECT — это формирование списка выражений, которые будут выведены в качестве итоговых данных. Для формирования номера места используется операция конкатенации «||», которая соединяет номер ряда с буквенным обозначением позиции в ряду.

```
SELECT aircraft_code, seat_row || letter, fare_condition
```

Итак, SQL-команда, которая позволит за одну операцию ввести в таблицу «Места» сразу необходимое число строк, выглядит так:

```
INSERT INTO seats ( aircraft_code, seat_no, fare_conditions )
-- номер места формируется с помощью конкатенации
-- номера ряда и буквы, обозначающей позицию в ряду
  SELECT aircraft_code, seat_row || letter, fare_condition
-- сформируем виртуальную таблицу для всех типов самолетов
-- колонки такие: код самолета, максимальные номера рядов кресел
-- в бизнес-классе и в экономическом классе, максимальный
-- номер кресла в ряду (он обозначается латинской буквой)
    FROM (
      VALUES ( 'SU9', 3, 20, 'F' ),
             ( '773', 5, 30, 'I' ),
             ( '763', 4, 25, 'H' ),
             ( '733', 3, 20, 'F' ),
             ( '320', 5, 25, 'F' ),
             ( '321', 4, 20, 'F' ),
             ( '319', 3, 20, 'F' ),
             ( 'CN1', 0, 10, 'B' ),
             ( 'CR2', 2, 15, 'D' )
```

```
    ) AS aircraft_info (
         aircraft_code, max_seat_row_business,
         max_seat_row_economy, max_letter
    )
  CROSS JOIN
-- классы обслуживания
    ( VALUES ( 'Business' ), ( 'Economy' ))
    AS fare_conditions ( fare_condition )
  CROSS JOIN
-- список номеров рядов кресел
    ( VALUES ( '1' ), ( '2' ), ( '3' ), ( '4' ), ( '5' ),
             ( '6' ), ( '7' ), ( '8' ), ( '9' ), ( '10' ),
             ( '11' ), ( '12' ), ( '13' ), ( '14' ), ( '15' ),
             ( '16' ), ( '17' ), ( '18' ), ( '19' ), ( '20' ),
             ( '21' ), ( '22' ), ( '23' ), ( '24' ), ( '25' ),
             ( '26' ), ( '27' ), ( '28' ), ( '29' ), ( '30' )
    ) AS seat_rows ( seat_row )
  CROSS JOIN
-- список номеров (позиций) кресел в ряду
    ( VALUES ( 'A' ), ( 'B' ), ( 'C' ), ( 'D' ), ( 'E' ),
             ( 'F' ), ( 'G' ), ( 'H' ), ( 'I' )
    ) AS letters ( letter )
-- включаем в результат лишь те строки, у которых номер ряда
-- (число) и позиция в ряду (латинская буква) соответствуют
-- диапазонам, указанным в первой виртуальной таблице
  WHERE
    CASE WHEN fare_condition = 'Business'
           THEN seat_row::integer <= max_seat_row_business
         WHEN fare_condition = 'Economy'
           THEN seat_row::integer > max_seat_row_business
            AND seat_row::integer <= max_seat_row_economy
    END
    AND letter <= max_letter;
```

**Задание:** модифицируйте команду с учетом того, что в салоне бизнес-класса число мест в ряду должно быть меньше, чем в салоне экономического класса (в приведенном решении мы для упрощения задачи принимали эти числа одинаковыми).

Попробуйте упростить подзапрос, отвечающий за формирование списка номеров рядов кресел:

```
( VALUES ( '1' ), ( '2' ), ( '3' ), ( '4' ), ( '5' ), ...
```

Воспользуйтесь функцией generate_series, описанной в разделе документации 9.24 «Функции, возвращающие множества».

# 8 Индексы

Индексы позволяют повысить производительность базы данных. PostgreSQL поддерживает различные типы индексов. Мы ограничимся рассмотрением только индексов на основе B-дерева.

Индекс — это специальная структура данных, которая связана с таблицей и создается на основе данных, содержащихся в ней. Основная цель создания индексов — повышение производительности функционирования базы данных.

## 8.1 Общая информация

Строки в таблицах хранятся в неупорядоченном виде. При выполнении операций выборки, обновления и удаления СУБД должна отыскать нужные строки. Для ускорения этого поиска и создается индекс. В принципе он организован таким образом: на основе данных, содержащихся в конкретной строке таблицы, формируется значение элемента (записи) индекса, соответствующего этой строке. Для поддержания соответствия между элементом индекса и строкой таблицы в каждый элемент помещается указатель на строку. Индекс является упорядоченной структурой. Элементы (записи) в нем хранятся в отсортированном виде, что значительно ускоряет поиск данных в индексе. После отыскания в нем требуемой записи СУБД переходит к соответствующей строке таблицы по прямой ссылке. Записи индекса могут формироваться на основе значений одного или нескольких полей соответствующих строк таблицы. Значения этих полей могут комбинироваться и преобразовываться различными способами. Все это определяет разработчик базы данных при создании индекса.

При выполнении поиска конкретных строк в таблице специальная подсистема СУБД, называемая планировщиком, проверяет, имеется ли для этой таблицы индекс, созданный на основе тех же столбцов, что указаны, например, в условии предложения WHERE. Если такой индекс существует, то планировщик оценивает целесообразность его использования в данном конкретном случае. Если его использование целесообразно, то сначала выполняется поиск необходимых значений в индексе, а затем, если такие значения в нем найдены, производится обращение к таблице с использованием указателей, которые хранятся в записях индекса. Таким образом, полный перебор строк в таблице может быть заменен поиском в упорядоченном индексе и переходом к строке таблицы по прямому указателю (ссылке).

Следует учитывать, что индексы требуют и некоторых накладных расходов на их создание и поддержание в актуальном состоянии при выполнении обновлений данных в таблицах. Поэтому использовать индексы нужно осмотрительно.

Когда вы создавали таблицы, то видели, что, как правило, для них предусматривалось создание первичного ключа — PRIMARY KEY. В таких случаях СУБД сама создает индекс, который позволяет поддерживать реализацию этого ограничения. Ведь при наличии первичного ключа не допускается появление в таблице строк с одинаковыми его значениями. Индекс позволяет выполнять проверку на дублирование очень быстро.

Для некоторых таблиц, например, «Посадочные талоны» (boarding_passes), было предусмотрено и ограничение уникальности UNIQUE. В этих случаях СУБД также автоматически создает индекс, который используется для обеспечения уникальности значений.

Для того чтобы увидеть индексы, созданные для данной таблицы, нужно воспользоваться командой утилиты psql:

```
\d имя_таблицы
```

Например,

```
\d boarding_passes
```

```
...
Индексы:
"boarding_passes_pkey" PRIMARY KEY, btree (ticket_no, flight_id)
"boarding_passes_flight_id_boarding_no_key" UNIQUE CONSTRAINT, btree
 ↪  (flight_id, boarding_no)
"boarding_passes_flight_id_seat_no_key" UNIQUE CONSTRAINT, btree
 ↪  (flight_id, seat_no)
...
```

Каждый индекс, который был создан самой СУБД, имеет типовое имя, состоящее из следующих компонентов:

- имени таблицы и суффикса pkey — для первичного ключа;
- имени таблицы, имен столбцов, по которым создан индекс, и суффикса key — для уникального ключа.

В описании также присутствует список столбцов, по которым создан индекс, и тип индекса — в данном случае это btree, т. е. B-дерево. PostgreSQL может создавать индексы различных типов, но по умолчанию используется так называемое B-дерево. Такой индекс подходит для большинства типовых задач. В этой главе мы будем рассматривать только индексы на основе B-дерева.

Наличие индекса может ускорить выборку строк из таблицы, если он создан по столбцам, на основе значений которых и производится выборка. Поэтому, как правило, при разработке и эксплуатации баз данных не ограничиваются только индексами, которые автоматически создает СУБД, а создают дополнительные индексы с учетом наиболее часто выполняющихся выборок.

Для создания индексов предназначена команда

```
CREATE INDEX имя_индекса ON имя_таблицы ( имя_столбца, ...);
```

В этой команде имя индекса можно не указывать. В качестве примера давайте создадим индекс для таблицы «Аэропорты» (airports) по столбцу airport_name.

```
CREATE INDEX ON airports ( airport_name );
```

```
CREATE INDEX
```

Посмотрим описание нового индекса:

```
\d airports
```

```
...
Индексы:
...
"airports_airport_name_idx" btree (airport_name)
...
```

Обратите внимание, что имя индекса, сформированное автоматически, включает имя таблицы, имя столбца и суффикс idx.

Прежде чем приступить к экспериментам с индексами, нужно включить в утилите psql секундомер с помощью следующей команды:

```
\timing on
```

Когда необходимость в использовании секундомера отпадет, для его отключения нужно будет сделать так:

```
\timing off
```

Теперь psql будет сообщать время, затраченное на выполнение всех команд.

Для практической проверки влияния индекса на скорость выполнения выборок сначала выполним следующий запрос:

```
SELECT count( * ) FROM tickets
  WHERE passenger_name = 'IVAN IVANOV';
```

```
 count
-------
   200
(1 строка)

Время: 373,232 мс
```

Показатели времени, полученные на вашем компьютере, конечно, будут отличаться от значений, приведенных в книге, и — возможно — значительно. Эти показатели нужно рассматривать лишь как качественные ориентиры.

Создадим индекс по столбцу passenger_name, при этом никакого суффикса в имени индекса использовать не будем, поскольку его наличие не является обязательным:

```
CREATE INDEX passenger_name
  ON tickets ( passenger_name );
```

```
CREATE INDEX
Время: 4023,408 мс
```

Посмотрим описание нового индекса:

```
\d tickets
```

```
...
Индексы:
...
"passenger_name" btree (passenger_name)
```

Теперь выполним ту же выборку из таблицы tickets:

```
SELECT count( * ) FROM tickets
  WHERE passenger_name = 'IVAN IVANOV';

 count
-------
   200
(1 строка)

Время: 17,660 мс
```

Вы видите, что время выполнения выборки при наличии индекса оказалось значительно меньше.

Просмотреть список всех индексов в текущей базе данных можно с помощью команды

```
\di
```

или

```
\di+
```

Для удаления индекса используется команда:

```
DROP INDEX имя_индекса;
```

Давайте удалим созданный нами индекс для таблицы tickets:

```
DROP INDEX passenger_name;

DROP INDEX
```

Когда индекс уже создан, о его поддержании в актуальном состоянии заботится СУБД. Конечно, следует учитывать, что это требует от СУБД затрат ресурсов и времени. Индекс, созданный по столбцу, участвующему в соединении двух таблиц, может позволить ускорить процесс выборки записей из таблиц. При выборке записей в отсортированном порядке индекс также может помочь, если сортировка выполняется по тем столбцам, по которым индекс создан.

## 8.2 Индексы по нескольким столбцам

Индексы могут создаваться не только по одному столбцу, но и по нескольким. Например, индекс для поддержания первичного ключа таблицы «Перелеты» (ticket_flights) создан по двум столбцам: ticket_no и flight_id.

Если в SQL-запросе есть предложение ORDER BY, то индекс может позволить избежать этапа сортировки выбранных строк. Однако если SQL-запрос просматривает значительную часть таблицы, то явная сортировка выбранных строк может оказаться быстрее, чем использование индекса. Создавая индексы с целью ускорения доступа к данным, нужно учитывать предполагаемую долю строк таблицы (селективность), выбираемых при выполнении типичных запросов, в которых создаваемый индекс будет использоваться. Если эта доля велика (т. е. селективность — низкая), тогда наличие индекса может не дать ожидаемого эффекта. Индексы более полезны, когда

из таблицы выбирается лишь небольшая доля строк, т. е. при *высокой селективности* выборки. В случае использования предложения ORDER BY в комбинации с LIMIT *n* явная сортировка (при отсутствии индекса) потребует обработки всех строк таблицы ради того, чтобы определить первые *n* строк. Но если есть индекс по тем же столбцам, по которым производится сортировка ORDER BY, то эти первые *n* строк могут быть извлечены непосредственно, без сканирования остальных строк вообще.

Если для таблицы «Билеты» (tickets) еще не создан индекс по столбцу book_ref, то создайте его:

```
CREATE INDEX tickets_book_ref_test_key
  ON tickets ( book_ref );
```

```
CREATE INDEX
```

Выполните запрос, в котором используется предложение LIMIT:

```
SELECT * FROM tickets ORDER BY book_ref LIMIT 5;
```

```
...
Время: 0,442 мс
```

Удалите этот индекс и повторите запрос. Время его выполнения увеличится, вероятно, на два порядка.

При создании индексов может использоваться не только возрастающий порядок значений в индексируемом столбце, но также и убывающий. По умолчанию порядок возрастающий, при этом значения NULL, которые также могут присутствовать в индексируемых столбцах, идут последними. При создании индекса можно модифицировать поведение по умолчанию с помощью ключевых слов ASC (возрастающий порядок), DESC (убывающий порядок), NULLS FIRST (эти значения идут первыми) и NULLS LAST (эти значения идут последними). Например:

```
CREATE INDEX имя_индекса
  ON имя_таблицы
  ( имя_столбца NULLS FIRST, ... );

CREATE INDEX имя_индекса
  ON имя_таблицы
  ( имя_столбца DESC NULLS LAST, ... );
```

## 8.3 Уникальные индексы

Индексы могут также использоваться для обеспечения уникальности значений атрибутов в строках таблицы. В таком случае создается уникальный индекс. Для его создания используется команда:

```
CREATE UNIQUE INDEX имя_индекса
  ON имя_таблицы
  ( имя_столбца, ...);
```

Например, создадим уникальный индекс по столбцу model для таблицы «Самолеты» (aircrafts):

```
CREATE UNIQUE INDEX aircrafts_unique_model_key
  ON aircrafts ( model );
```

В этом случае мы уже не сможем ввести в таблицу aircrafts строки, имеющие одинаковые наименования моделей самолетов. Конечно, мы могли при создании таблицы задать ограничение уникальности для столбца model, и тогда уникальный индекс был бы создан автоматически.

Важно, что в уникальных индексах допускается наличие значений NULL, поскольку они считаются не совпадающими ни с какими другими значениями, в том числе и друг с другом. Если уникальный индекс создан по нескольким атрибутам, то совпадающими считаются лишь те комбинации значений атрибутов в двух строках, в которых совпадают значения всех соответственных атрибутов.

## 8.4 Индексы на основе выражений

В команде создания индекса можно использовать не только имена столбцов, но также функции и скалярные выражения, построенные на основе столбцов таблицы. Например, если мы захотим запретить значения столбца model в таблице aircrafts, отличающиеся только регистром символов, то создадим такой индекс:

```
CREATE UNIQUE INDEX aircrafts_unique_model_key
  ON aircrafts ( lower( model ) );
```

Если теперь попытаться добавить строку, в которой значение атрибута model будет «Cessna 208 CARAVAN», то PostgreSQL выдаст сообщение об ошибке, даже если значение атрибута aircraft_code будет уникальным.

```
INSERT INTO aircrafts VALUES ( '123', 'Cessna 208 CARAVAN', 1300);
```

```
ОШИБКА: повторяющееся значение ключа нарушает ограничение уникальности
 ↪  "aircrafts_unique_model_key"
ПОДРОБНОСТИ: Ключ "(lower(model))=(cessna 208 caravan)" уже существует.
```

Встроенная функция lower преобразует все символы в нижний регистр. Индекс строится уже на основе преобразованных значений, поэтому при поиске строки в таблице искомое значение сначала переводится в нижний регистр, а затем осуществляется поиск в индексе.

Индексы на основе выражений требуют больше ресурсов для их создания и поддержания при вставке и обновлении строк в таблице. Зато при выполнении выборок, построенных на основе сложных выражений, работа происходит с меньшими накладными расходами, поскольку в индексе хранятся уже вычисленные значения этих выражений, пусть даже самых сложных. Поэтому такие индексы целесообразно использовать тогда, когда выборки производятся многократно, и время, затраченное на создание и поддержание индекса, компенсируется (окупается) при выполнении выборок из таблицы.

## 8.5 Частичные индексы

PostgreSQL поддерживает очень интересный тип индексов — **частичные индексы**. Такой индекс формируется не для всех строк таблицы, а лишь для их подмножества. Это достигается с помощью использования условного выражения, называемого *предикатом индекса*. Предикат вводится с помощью предложения WHERE.

В качестве иллюстрации создадим частичный индекс для таблицы «Бронирования» (bookings). Представим, что руководство компании интересуют бронирования на сумму свыше одного миллиона рублей. Такая выборка выполняется с помощью запроса

```
SELECT * FROM bookings
  WHERE total_amount > 1000000
  ORDER BY book_date DESC;
```

```
 book_ref |       book_date        | total_amount
----------+------------------------+--------------
 D7E9AA   | 2016-10-06 09:29:00+08 |  1062800.00
 EF479E   | 2016-09-30 19:58:00+08 |  1035100.00
 3AC131   | 2016-09-28 05:06:00+08 |  1087100.00
 3B54BB   | 2016-09-02 21:08:00+08 |  1204500.00
 65A6EA   | 2016-08-31 10:28:00+08 |  1065600.00
(5 строк)

Время: 90,996 мс
```

Хотя сортировка строк производится по датам бронирования в убывающем порядке, т. е. от более поздних дат к более ранним, тем не менее, включать ключевое слово DESC в индексное выражение, когда индекс создается только по одному столбцу, нет необходимости. Это объясняется тем, что PostgreSQL умеет совершать обход индекса как по возрастанию, так и по убыванию с одинаковой эффективностью.

Обратите внимание, что индексируемый столбец book_date не участвует в формировании предиката индекса — в предикате используется столбец total_amount. Это вполне допустимая ситуация.

```
CREATE INDEX bookings_book_date_part_key
  ON bookings ( book_date )
  WHERE total_amount > 1000000;
```

```
CREATE INDEX
```

Повторим вышеприведенный запрос. Теперь он выдаст результат за время, на порядок меньшее, чем без использования частичного индекса.

В разделе документации 11.8 «Частичные индексы» сказано, что для того чтобы СУБД использовала частичный индекс, необходимо чтобы условие, записанное в запросе в предложении WHERE, соответствовало предикату индекса. Это означает, что либо условие должно быть точно таким же, как использованное в предикате частичного индекса при его создании, либо условие запроса должно математически сводиться к предикату индекса, а система должна суметь это понять. Например, в таком запросе индекс будет использоваться:

```
SELECT * FROM bookings WHERE total_amount > 1100000 ...
```

А в таком не будет:

```
SELECT * FROM bookings WHERE total_amount > 900000 ...
```

Частичные индексы выглядят очень привлекательно, но в большинстве случаев их преимущества по сравнению с обычными индексами будут минимальными (см. задание 9). Однако размер частичного индекса будет меньше, чем размер обычного. Для получения заметного полезного эффекта от их применения необходим опыт и понимание того, как работают индексы в PostgreSQL.

## Контрольные вопросы и задания

1. Предположим, что для какой-то таблицы создан уникальный индекс по двум столбцам: column1 и column2. В таблице есть строка, у которой значение атрибута column1 равно «ABC», а значение атрибута column2 — NULL. Мы решили добавить в таблицу еще одну строку с такими же значениями ключевых атрибутов, т. е. column1 — «ABC», а column2 — NULL.

   Как вы думаете, будет ли операция вставки новой строки успешной или завершится с ошибкой? Объясните ваше решение.

2. В тексте главы шла речь о выполнении одной и той же выборки из таблицы «Билеты» (tickets) при наличии индекса по столбцу passenger_name и при его отсутствии. Вы видели, что наличие индекса ускоряет выполнение запроса почти на порядок.

   Если секундомер в утилите psql выключен, то включите его с помощью команды

   ```
   \timing on
   ```

   Проведите следующий эксперимент: выполните этот запрос несколько раз подряд при отсутствии индекса, а затем создайте индекс и опять выполните этот запрос несколько раз подряд.

   ```
   SELECT count( * ) FROM tickets
     WHERE passenger_name = 'IVAN IVANOV';
   ```

   Вы увидите, что время выполнения *повторных* запросов к таблице сокращается, причем, когда создан индекс, оно сокращается на порядок. Как вы думаете, почему?

3. Известно, что индекс значительно ускоряет работу, если при выполнении запроса из таблицы отбирается лишь небольшая часть строк. Если же эта доля велика, скажем, половина строк или более, то большого положительного эффекта от наличия индекса уже не будет, а возможно даже, что не будет практически никакого эффекта. Наша задача — проверить это утверждение на практике.

   Обратимся к таблице «Перелеты» (ticket_flights). В ней есть столбец «Класс обслуживания» (fare_conditions). Этот столбец отличается тем, что в нем могут присутствовать лишь три различных значения: «Comfort», «Business» и «Economy».

   Если секундомер в утилите psql выключен, то включите его.

Выполните запросы, подсчитывающие количество строк, в которых атрибут fare_conditions принимает одно из трех возможных значений. Каждый из запросов выполните три-четыре раза, поскольку время может немного изменяться, и подсчитайте среднее время. Обратите внимание на число строк, возвращаемое функцией count для каждого значения атрибута fare_conditions. При этом среднее время выполнения запросов для трех различных значений атрибута fare_conditions будет различаться незначительно, поскольку в каждом случае СУБД просматривает все строки таблицы.

```
SELECT count( * ) FROM ticket_flights
  WHERE fare_conditions = 'Comfort';

SELECT count( * ) FROM ticket_flights
  WHERE fare_conditions = 'Business';

SELECT count( * ) FROM ticket_flights
  WHERE fare_conditions = 'Economy';
```

Создайте индекс по столбцу fare_conditions. Конечно, в реальной ситуации такой индекс вряд ли целесообразно создавать, но нам он нужен для экспериментов.

Проделайте те же эксперименты с таблицей ticket_flights. Будет ли различаться среднее время выполнения запросов для различных значений атрибута fare_conditions? Почему это имеет место?

В завершение этого упражнения отметим, что в случае ошибки планировщика при использовании индекса возможно не только отсутствие положительного эффекта, но и значительный отрицательный эффект.

4. Для одной из таблиц создайте индекс по двум столбцам, причем по одному из них укажите убывающий порядок значений столбца, а по другому — возрастающий. Значения NULL у первого столбца должны располагаться в начале, а у второго — в конце. Посмотрите полученный индекс с помощью команд утилиты psql

```
\d имя_таблицы
\di+ имя_индекса
```

Обратите внимание, что первая команда выведет не только имя индекса, но также и имена столбцов, по которым он создан, а вторая команда выведет размер индекса.

Подберите запросы, в которых созданный индекс предположительно должен использоваться, а также запросы, в которых он использоваться, по вашему мнению, не будет. Проверьте ваши гипотезы, выполнив запросы. Объясните полученные результаты.

5. В сложных базах данных целесообразно использование комбинаций индексов. Иногда бывают более полезны комбинированные индексы по нескольким столбцам, чем отдельные индексы по единичным столбцам. В реальных ситуациях часто приходится делать выбор, т. е. находить компромисс, между, например, созданием двух индексов по каждому из двух столбцов таблицы либо созданием одного индекса по двум столбцам этой таблицы, либо созданием всех трех индексов. Выбор зависит от того, запросы какого вида будут выполняться

чаще всего. Предложите какую-нибудь таблицу в базе данных «Авиаперевозки» и смоделируйте ситуации, в которых вы приняли бы одно из этих трех возможных решений. Воспользуйтесь документацией на PostgreSQL.

6. Предложите какую-нибудь таблицу в базе данных «Авиаперевозки» и смоделируйте ситуацию, в которой было бы целесообразно использование индекса на основе функции или скалярного выражения от двух или более столбцов.

7.* В разделе документации 5.3.5 «Внешние ключи» говорится о том, что в некоторых ситуациях бывает целесообразно создавать индекс по столбцам внешнего ключа ссылающейся таблицы. Это позволит ускорить выполнение операций DELETE и UPDATE над главной (ссылочной) таблицей.

   Подумайте, есть ли такие таблицы в базе данных «Авиаперевозки», в отношении которых было бы целесообразно поступить так, как говорится в документации.

8.* В тексте главы был показан пример использования частичного индекса для таблицы «Бронирования» (bookings). Для его создания мы выполняли команду

```
CREATE INDEX bookings_book_date_part_key ON bookings ( book_date )
WHERE total_amount > 1000000;
```

   Проведите эксперимент с целью сравнения эффекта от создания частичного индекса с эффектом от создания обычного индекса по столбцу total_amount. Для этого удалите частичный индекс, а затем создайте обычный индекс.

```
DROP INDEX bookings_book_date_part_key;

CREATE INDEX bookings_total_amount_key
  ON bookings ( total_amount );
```

   Теперь выполните тот же запрос к таблице bookings, который был приведен в тексте главы:

```
SELECT * FROM bookings
  WHERE total_amount > 1000000
  ORDER BY book_date DESC;
```

   Сравните время выполнения с тем временем, которое было получено при использовании частичного индекса. Очень вероятно, что различия времени выполнения запроса будут незначительными.

   Самостоятельно ознакомьтесь с разделом документации 11.8 «Частичные индексы» и попробуйте смоделировать ситуацию в предметной области «Авиаперевозки», когда частичный индекс дал бы больший эффект, чем обычный индекс.

9. Когда выполняются запросы с поиском по шаблону LIKE или регулярными выражениями POSIX, тогда для того, чтобы использовался индекс, нужно предусмотреть следующее. Если параметры локализации системы отличаются от стандартной настройки «C» (например, «ru_RU.UTF-8»), тогда при создании индекса необходимо указать так называемый класс операторов. Существуют различные классы, например, для столбца типа text это будет text_pattern_ops.

```
CREATE INDEX tickets_pass_name
  ON tickets ( passenger_name text_pattern_ops );
```

Индексы со специальными классами операторов пригодны не для всех типов запросов. Поэтому, возможно, потребуется создать еще и индекс с классом операторов по умолчанию. Самостоятельно изучите этот вопрос с помощью раздела документации 11.9 «Семейства и классы операторов».

# 9 Транзакции

Детальное понимание механизмов выполнения транзакций придет с опытом. В этом разделе мы дадим самое первое представление об этом важном и мощном инструменте, которым обладают все серьезные СУБД, включая PostgreSQL.

Транзакция — это совокупность операций над базой данных, которые вместе образуют логически целостную процедуру, и могут быть либо выполнены все вместе, либо не будет выполнена ни одна из них. В простейшем случае транзакция состоит из одной операции.

Транзакции являются одним из средств обеспечения согласованности (непротиворечивости) базы данных, наряду с ограничениями целостности (constraints), накладываемыми на таблицы. Транзакция переводит базу данных из одного согласованного состояния в другое согласованное состояние. В качестве примера транзакции в базе данных «Авиаперевозки» можно привести процедуру бронирования билета. Она будет включать операции INSERT, выполняемые над таблицами «Бронирования» (bookings), «Билеты» (tickets) и «Перелеты» (ticket_flights). В результате выполнения этой транзакции должно обеспечиваться следующее соотношение: значение атрибута total_amount в строке таблицы bookings должно быть равно сумме значений атрибута amount в строках таблицы ticket_flights, связанных с этой строкой таблицы bookings. Если операции данной транзакции будут выполнены частично, тогда может оказаться, например, что общая сумма бронирования будет не равна сумме стоимостей перелетов, включенных в это бронирование. Очевидно, что это несогласованное состояние базы данных.

Транзакция может иметь два исхода: первый — изменения данных, произведенные в ходе ее выполнения, успешно зафиксированы в базе данных, а второй исход таков — транзакция отменяется, и отменяются все изменения, выполненные в ее рамках. Отмена транзакции называется откатом (rollback).

Сложные информационные системы, как правило, предполагают одновременную работу многих пользователей с базой данных, поэтому современные СУБД предлагают специальные механизмы для организации параллельного, т. е. одновременного, выполнения транзакций. Реализованы такие механизмы и в PostgreSQL.

Реализация транзакций в СУБД PostgreSQL основана на многоверсионной модели (Multiversion Concurrency Control, MVCC). Эта модель предполагает, что каждый SQL-оператор видит так называемый *снимок* данных (snapshot), т. е. то согласованное состояние (версию) базы данных, которое она имела на определенный момент времени. При этом параллельно исполняемые транзакции, даже вносящие изменения в базу данных, не нарушают согласованности данных этого снимка. Такой результат в PostgreSQL достигается за счет того, что когда параллельные транзакции изменяют одни и те же строки таблиц, тогда создаются отдельные *версии* этих строк, доступные соответствующим транзакциям. Это позволяет ускорить работу с базой данных, однако требует больше дискового пространства и оперативной памяти. И еще одно важное следствие применения MVCC — операции чтения никогда не блокируются операциями записи, а операции записи никогда не блокируются операциями чтения.

Согласно теории баз данных, транзакции должны обладать следующими свойствами:

1. Атомарность (atomicity). Это свойство означает, что либо транзакция будет зафиксирована в базе данных полностью, т. е. будут зафиксированы результаты выполнения всех ее операций, либо не будет зафиксирована ни одна операция транзакции.
2. Согласованность (consistency). Это свойство предписывает, чтобы в результате успешного выполнения транзакции база данных была переведена из одного согласованного состояния в другое согласованное состояние.
3. Изолированность (isolation). Во время выполнения транзакции другие транзакции должны оказывать по возможности минимальное влияние на нее.
4. Долговечность (durability). После успешной фиксации транзакции пользователь должен быть уверен, что данные надежно сохранены в базе данных и впоследствии могут быть извлечены из нее, независимо от последующих возможных сбоев в работе системы.

Для обозначения всех этих четырех свойств используется аббревиатура ACID.

При параллельном выполнении транзакций теоретически возможны следующие феномены.

1. Потерянное обновление (lost update). Когда разные транзакции одновременно изменяют одни и те же данные, то после фиксации изменений может оказаться, что одна транзакция перезаписала данные, обновленные и зафиксированные другой транзакцией.
2. «Грязное» чтение (dirty read). Транзакция читает данные, измененные параллельной транзакцией, которая еще не завершилась. Если эта параллельная транзакция в итоге будет отменена, тогда окажется, что первая транзакция прочитала данные, которых нет в системе.
3. Неповторяющееся чтение (non-repeatable read). При повторном чтении тех же самых данных в рамках одной транзакции оказывается, что другая транзакция успела изменить и зафиксировать эти данные. В результате тот же самый запрос выдает другой результат.
4. Фантомное чтение (phantom read). Транзакция выполняет повторную выборку множества строк в соответствии с одним и тем же критерием. В интервале времени между выполнением этих выборок другая транзакция добавляет новые строки и успешно фиксирует изменения. В результате при выполнении повторной выборки в первой транзакции может быть получено другое множество строк.
5. Аномалия сериализации (serialization anomaly). Результат успешной фиксации группы транзакций, выполняющихся параллельно, не совпадает с результатом ни одного из возможных вариантов упорядочения этих транзакций, если бы они выполнялись последовательно.

Перечисленные феномены, а также ситуации, в которых они имеют место, будут рассмотрены подробно и проиллюстрированы примерами.

Поясним кратко, в чем состоит смысл концепции сериализации. Для двух транзакций, скажем, A и B, возможны только два варианта упорядочения при их последовательном выполнении: сначала A, затем B или сначала B, затем A. Причем результаты

реализации двух вариантов могут в общем случае не совпадать. Например, при выполнении двух банковских операций — внесения некоторой суммы денег на какой-то счет и начисления процентов по этому счету — важен порядок выполнения операций. Если первой операцией будет увеличение суммы на счете, а второй — начисление процентов, тогда итоговая сумма будет больше, чем при противоположном порядке выполнения этих операций. Если описанные операции выполняются в рамках двух различных транзакций, то оказываются возможными различные итоговые результаты, зависящие от порядка их выполнения.

Сериализация двух транзакций при их *параллельном* выполнении означает, что полученный результат будет соответствовать *одному* из двух возможных вариантов упорядочения транзакций при их последовательном выполнении. При этом нельзя сказать точно, какой из вариантов будет реализован.

Если распространить эти рассуждения на случай, когда параллельно выполняется более двух транзакций, тогда результат их параллельного выполнения также должен быть таким, каким он был бы в случае выбора *некоторого варианта* упорядочения транзакций, если бы они выполнялись последовательно, одна за другой. Конечно, чем больше транзакций, тем больше вариантов их упорядочения. Концепция сериализации не предписывает выбора какого-то определенного варианта. Речь идет лишь об одном из них.

В том случае, если СУБД не сможет гарантировать успешную сериализацию группы параллельных транзакций, тогда некоторые из них могут быть завершены с ошибкой. Эти транзакции придется выполнить повторно.

Для конкретизации степени независимости параллельных транзакций вводится понятие уровня *изоляции транзакций*. Каждый уровень характеризуется перечнем тех феноменов, которые на данном уровне не допускаются.

Всего в стандарте SQL предусмотрено четыре уровня. Каждый более высокий уровень включает в себя все возможности предыдущего.

1. READ UNCOMMITTED. Это самый низкий уровень изоляции. Согласно стандарту SQL, на этом уровне допускается чтение «грязных» (незафиксированных) данных. Однако в PostgreSQL требования, предъявляемые к этому уровню, более строгие, чем в стандарте: чтение «грязных» данных на этом уровне не допускается.
2. READ COMMITTED. Не допускается чтение «грязных» (незафиксированных) данных. Таким образом, в PostgreSQL уровень READ UNCOMMITTED совпадает с уровнем READ COMMITTED. Транзакция может видеть только те незафиксированные изменения данных, которые произведены в ходе выполнения ее самой.
3. REPEATABLE READ. Не допускается чтение «грязных» (незафиксированных) данных и неповторяющееся чтение. В PostgreSQL на этом уровне не допускается также фантомное чтение. Таким образом, реализация этого уровня является более строгой, чем того требует стандарт SQL. Это не противоречит стандарту.
4. SERIALIZABLE. Не допускается ни один из феноменов, перечисленных выше, в том числе и аномалии сериализации.

Конкретный уровень изоляции обеспечивает сама СУБД с помощью своих внутренних механизмов. Его достаточно указать в команде при старте транзакции. Однако программист может дополнительно использовать некоторые операторы и приемы программирования, например, устанавливать блокировки на уровне отдельных строк или всей таблицы. Это будет показано в конце главы.

По умолчанию PostgreSQL использует уровень изоляции READ COMMITTED.

```
SHOW default_transaction_isolation;
```

```
 default_transaction_isolation
-------------------------------
 read committed
(1 строка)
```

## 9.1 Уровень изоляции READ UNCOMMITTED

Давайте начнем рассмотрение с уровня изоляции READ UNCOMMITTED. Проверим, видит ли транзакция те изменения данных, которые были произведены в другой транзакции, но еще не были зафиксированы, т. е. «грязные» данные.

Для проведения экспериментов воспользуемся таблицей «Самолеты» (aircrafts). Но можно создать копию этой таблицы, чтобы при удалении строк из нее не удалялись строки из таблицы «Места» (seats), связанные со строками из таблицы aircrafts по внешнему ключу.

```
CREATE TABLE aircrafts_tmp AS SELECT * FROM aircrafts;
```

```
SELECT 9
```

Для организации выполнения параллельных транзакций с использованием утилиты psql будем запускать ее на двух терминалах.

Итак, для изучения уровня изоляции READ UNCOMMITTED проделаем следующие эксперименты.

На первом терминале выполним следующие команды:

```
BEGIN;
```

```
BEGIN
```

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

```
SET
```

```
SHOW transaction_isolation;
```

```
 transaction_isolation
-----------------------
 read uncommitted
(1 строка)
```

```
UPDATE aircrafts_tmp
  SET range = range + 100
  WHERE aircraft_code = 'SU9';
```

```
UPDATE 1
```

```
SELECT * FROM aircrafts_tmp WHERE aircraft_code = 'SU9';
```

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 SU9           | Sukhoi SuperJet-100 | 3100
(1 строка)
```

Начнем транзакцию на втором терминале (все, что происходит на втором терминале, показано на сером фоне):

```
BEGIN;
```

```
BEGIN
```

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

```
SELECT * FROM aircrafts_tmp WHERE aircraft_code = 'SU9';
```

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 SU9           | Sukhoi SuperJet-100 |  3000
(1 строка)
```

Таким образом, вторая транзакция не видит изменение значения атрибута range, произведенное в первой — незафиксированной — транзакции. Это объясняется тем, что в PostgreSQL реализация уровня изоляции READ UNCOMMITTED более строгая, чем того требует стандарт языка SQL. Фактически этот уровень тождественен уровню изоляции READ COMMITTED. Поэтому будем считать эксперимент, проведенный для уровня изоляции READ UNCOMMITTED, выполненным и для уровня READ COMMITTED.

Давайте не будем фиксировать произведенное изменение в базе данных, а воспользуемся командой ROLLBACK для отмены транзакции, т. е. для ее отката.

На первом терминале:

```
ROLLBACK;
```

```
ROLLBACK
```

На втором терминале сделаем так же:

```
ROLLBACK;
```

```
ROLLBACK
```

## 9.2 Уровень изоляции READ COMMITTED

Теперь обратимся к уровню изоляции READ COMMITTED. Именно этот уровень установлен в PostgreSQL по умолчанию. Мы уже показали, что на этом уровне изоляции не допускается чтение незафиксированных данных. А сейчас покажем, что на этом уровне изоляции также гарантируется отсутствие потерянных обновлений, но возможно неповторяющееся чтение данных.

Опять будем работать на двух терминалах. В первой транзакции увеличим значение атрибута range для самолета Sukhoi SuperJet-100 на 100 км, а во второй транзакции — на 200 км. Проверим, какое из этих двух изменений будет записано в базу данных.

На первом терминале выполним следующие команды:

```
BEGIN ISOLATION LEVEL READ COMMITTED;
```

```
BEGIN
```

```
SHOW transaction_isolation;
```

```
 transaction_isolation
-----------------------
 read committed
(1 строка)
```

```
UPDATE aircrafts_tmp
  SET range = range + 100
  WHERE aircraft_code = 'SU9';
```

```
UPDATE 1
```

```
SELECT * FROM aircrafts_tmp WHERE aircraft_code = 'SU9';
```

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 SU9           | Sukhoi SuperJet-100 |  3100
(1 строка)
```

Мы видим, что в первой транзакции значение атрибута range было успешно изменено, хотя пока и не зафиксировано. Но транзакция видит изменения, выполненные в ней самой.

Обратите внимание, что вместо использования команды SET TRANSACTION мы просто включили указание уровня изоляции непосредственно в команду BEGIN. Эти два подхода равносильны. Конечно, когда речь идет об использовании уровня изоляции READ COMMITTED, принимаемого по умолчанию, можно вообще ограничиться только командой BEGIN без дополнительных ключевых слов.

На втором терминале так и сделаем. Во второй транзакции попытаемся обновить эту же строку таблицы airctafts_tmp, но для того, чтобы впоследствии разобраться, какое из изменений прошло успешно и было зафиксировано, добавим к значению атрибута range не 100, а 200.

```
BEGIN;
```

```
BEGIN
```

```
UPDATE aircrafts_tmp
  SET range = range + 200
  WHERE aircraft_code = 'SU9';
```

И вот мы видим, что команда UPDATE во второй транзакции не завершилась, а перешла в состояние ожидания. Это ожидание продлится до тех пор, пока не завершится первая транзакция. Дело в том, что команда UPDATE в первой транзакции заблокировала строку в таблице airctafts_tmp, и эта блокировка будет снята только при завершении транзакции либо с фиксацией изменений с помощью команды COMMIT, либо с отменой изменений по команде ROLLBACK.

Давайте завершим первую транзакцию с фиксацией изменений:

```
COMMIT;
```

```
COMMIT
```

Перейдя на второй терминал, мы увидим, что команда UPDATE завершилась:

```
UPDATE 1
```

Теперь на втором терминале, не завершая транзакцию, посмотрим, что стало с нашей строкой в таблице aircrafts_tmp:

```
SELECT * FROM aircrafts_tmp WHERE aircraft_code = 'SU9';
```

```
 aircraft_code |        model        | range
---------------+---------------------+-------
 SU9           | Sukhoi SuperJet-100 |  3300
(1 строка)
```

Как видно, были произведены оба изменения. Команда UPDATE во второй транзакции, получив возможность заблокировать строку после завершения первой транзакции и снятия ею блокировки с этой строки, *перечитывает* строку таблицы и потому обновляет строку, уже обновленную в только что зафиксированной транзакции. Таким образом, эффекта потерянных обновлений не возникает.

Завершим транзакцию на втором терминале, но при этом вместо команды COMMIT воспользуемся эквивалентной командой END, которая является расширением PostgreSQL:

```
END;
```

```
COMMIT
```

Если вы самостоятельно проведете только что выполненный эксперимент, выбрав уровень изоляции READ UNCOMMITTED, то увидите, что и на этом — самом низком — уровне изоляции эффекта потерянных обновлений также не возникает.

Для иллюстрации эффекта неповторяющегося чтения данных проведем совсем простой эксперимент также на двух терминалах.

На первом терминале:

```
BEGIN;
```

```
BEGIN
```

```
SELECT * FROM aircrafts_tmp;
```

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 773           | Boeing 777-300       | 11100
 763           | Boeing 767-300       |  7900
 320           | Airbus A320-200      |  5700
 321           | Airbus A321-200      |  5600
 319           | Airbus A319-100      |  6700
 733           | Boeing 737-300       |  4200
 CN1           | Cessna 208 Caravan   |  1200
 CR2           | Bombardier CRJ-200   |  2700
 SU9           | Sukhoi SuperJet-100  |  3700
(9 строк)
```

На втором терминале:

```
BEGIN;
```

```
BEGIN
```

```
DELETE FROM aircrafts_tmp WHERE model ~ '^Boe';
```

```
DELETE 3
```

```
SELECT * FROM aircrafts_tmp;
```

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 320           | Airbus A320-200      |  5700
 321           | Airbus A321-200      |  5600
 319           | Airbus A319-100      |  6700
 CN1           | Cessna 208 Caravan   |  1200
 CR2           | Bombardier CRJ-200   |  2700
 SU9           | Sukhoi SuperJet-100  |  3700
(6 строк)
```

Сразу завершим вторую транзакцию:

```
END;
```

```
COMMIT
```

Повторим выборку в первой транзакции:

```
SELECT * FROM aircrafts_tmp;
```

```
 aircraft_code |         model        | range
---------------+----------------------+-------
 320           | Airbus A320-200      |  5700
 321           | Airbus A321-200      |  5600
 319           | Airbus A319-100      |  6700
 CN1           | Cessna 208 Caravan   |  1200
 CR2           | Bombardier CRJ-200   |  2700
 SU9           | Sukhoi SuperJet-100  |  3700
(6 строк)
```

Видим, что теперь получен другой результат, т. к. вторая транзакция завершилась в момент времени между двумя запросами. Таким образом, налицо эффект неповторяющегося чтения данных, который является допустимым на уровне изоляции READ COMMITTED.

Завершим и первую транзакцию:

```
END;
```

```
COMMIT
```

## 9.3 Уровень изоляции REPEATABLE READ

Третий уровень изоляции — REPEATABLE READ. Само его название говорит о том, что он не допускает наличия феномена неповторяющегося чтения данных. А в PostgreSQL на этом уровне не допускается и чтение фантомных строк.

Приложения, использующие этот уровень изоляции должны быть готовы к тому, что придется выполнять транзакции повторно. Это объясняется тем, что транзакция, использующая этот уровень изоляции, создает снимок данных не перед выполнением каждого запроса, а только однократно, перед выполнением *первого запроса* транзакции. Поэтому транзакции с этим уровнем изоляции не могут изменять строки, которые были изменены другими завершившимися транзакциями уже после создания снимка. Вследствие этого PostgreSQL не позволит зафиксировать транзакцию, которая попытается изменить уже измененную строку.

Важно помнить, что повторный запуск может потребоваться только для транзакций, которые вносят изменения в данные. Для транзакций, которые только читают данные, повторный запуск никогда не требуется.

На первом терминале:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

Сначала посмотрим содержимое таблицы:

```
SELECT * FROM aircrafts_tmp;
```

Обратите внимание, что после уже проведенных экспериментов в таблице осталось меньше строк, чем было вначале.

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 320           | Airbus A320-200      |  5700
 321           | Airbus A321-200      |  5600
 319           | Airbus A319-100      |  6700
 SU9           | Sukhoi SuperJet-100 |  3700
 CN1           | Cessna 208 Caravan   |  2100
 CR2           | Bombardier CRJ-200   |  1900
(6 строк)
```

На втором терминале проведем ряд изменений:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

Добавим одну строку:

```
INSERT INTO aircrafts_tmp VALUES ( 'IL9', 'Ilyushin IL96', 9800 );
```

```
INSERT 0 1
```

А одну строку обновим:

```
UPDATE aircrafts_tmp SET range = range + 100
  WHERE aircraft_code = '320';
```

```
UPDATE 1
```

```
END;
```

```
COMMIT
```

Переходим на первый терминал.

```
SELECT * FROM aircrafts_tmp;
```

На первом терминале ничего не изменилось: фантомные строки не видны, и также не видны изменения в уже существующих строках. Это объясняется тем, что снимок данных выполняется на момент начала выполнения первого запроса транзакции.

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 320           | Airbus A320-200      |  5700
 321           | Airbus A321-200      |  5600
 319           | Airbus A319-100      |  6700
 SU9           | Sukhoi SuperJet-100  |  3700
 CN1           | Cessna 208 Caravan   |  2100
 CR2           | Bombardier CRJ-200   |  1900
(6 строк)
```

Завершим первую транзакцию тоже:

```
END;
```

```
COMMIT
```

А теперь посмотрим, что изменилось в таблице:

```
SELECT * FROM aircrafts_tmp;
```

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 321           | Airbus A321-200      |  5600
 319           | Airbus A319-100      |  6700
 SU9           | Sukhoi SuperJet-100  |  3700
 CN1           | Cessna 208 Caravan   |  2100
 CR2           | Bombardier CRJ-200   |  1900
 IL9           | Ilyushin IL96        |  9800
```

```
 320          | Airbus A320-200     |  5800
(7 строк)
```

Как видим, одна строка добавлена, а значение атрибута range у самолета Airbus A320-200 стало на 100 больше, чем было. Но до тех пор, пока мы на первом терминале находились в процессе выполнения первой транзакции, все эти изменения не были ей доступны, поскольку первая транзакция использовала снимок, сделанный до внесения изменений и их фиксации второй транзакцией.

Теперь покажем ошибки сериализации.

Начнем транзакцию на первом терминале:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

```
UPDATE aircrafts_tmp
  SET range = range + 100
  WHERE aircraft_code = '320';
```

```
UPDATE 1
```

На втором терминале попытаемся обновить ту же строку:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

```
UPDATE aircrafts_tmp
  SET range = range + 200
  WHERE aircraft_code = '320';
```

Команда UPDATE на втором терминале ожидает завершения первой транзакции.

Перейдя на первый терминал, завершим первую транзакцию:

```
END;
```

```
COMMIT
```

Перейдя на второй терминал, увидим сообщение об ошибке:

```
ОШИБКА: не удалось сериализовать доступ из-за параллельного изменения
```

Поскольку обновление, произведенное в первой транзакции, не было зафиксировано на момент начала выполнения первого (и, в данном частном случае, единственного) запроса во второй транзакции, то возникает эта ошибка. Это объясняется вот чем. При выполнении обновления строки команда UPDATE во второй транзакции видит, что строка уже изменена. На уровне изоляции REPEATABLE READ снимок данных создается на момент начала выполнения первого запроса транзакции и в течение транзакции уже не меняется, т. е. новая версия строки не считывается, как это делалось на уровне READ COMMITTED. Но если выполнить обновление во второй транзакции без повторного считывания строки из таблицы, тогда будет иметь место потерянное обновление, что недопустимо. В результате генерируется ошибка, и вторая транзакция откатывается. Мы вводим команду END на втором терминале, но PostgreSQL выполняет не фиксацию (COMMIT), а откат:

```
END;
```

```
ROLLBACK
```

Если выполним запрос, то увидим, что было проведено только изменение в первой транзакции:

```
SELECT * FROM aircrafts_tmp WHERE aircraft_code = '320';
```

```
 aircraft_code |        model     | range
---------------+------------------+-------
 320           | Airbus A320-200 |  5900
(1 строка)
```

## 9.4 Уровень изоляции SERIALIZABLE

Самый высший уровень изоляции транзакций — SERIALIZABLE. Транзакции могут работать параллельно точно так же, как если бы они выполнялись последовательно одна за другой. Однако, как и при использовании уровня REPEATABLE READ, приложение должно быть готово к тому, что придется перезапускать транзакцию, которая была прервана системой из-за обнаружения зависимостей чтения/записи между транзакциями. Группа транзакций может быть параллельно выполнена и успешно зафиксирована в том случае, когда результат их параллельного выполнения был бы эквивалентен результату выполнения этих транзакций при выборе *одного из возможных вариантов* их упорядочения, если бы они выполнялись последовательно, одна за другой.

Для проведения эксперимента создадим специальную таблицу, в которой будет всего два столбца: один — числовой, а второй — текстовый. Назовем эту таблицу — modes.

```
CREATE TABLE modes ( num integer, mode text );
```

```
CREATE TABLE
```

Добавим в таблицу две строки.

```
INSERT INTO modes VALUES ( 1, 'LOW' ), ( 2, 'HIGH' );
```

```
INSERT 0 2
```

Итак, содержимое таблицы имеет вид:

```
SELECT * FROM modes;
```

```
 num | mode
-----+------
   1 | LOW
   2 | HIGH
(2 строки)
```

На первом терминале начнем транзакцию и обновим одну строку из тех двух строк, которые были показаны в предыдущем запросе.

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN
```

В команде обновления строки будем использовать предложение RETURNING. Поскольку значение поля num не изменяется, то будет видно, какая строка была обновлена. Это особенно пригодится во второй транзакции.

```
UPDATE modes
  SET mode = 'HIGH'
  WHERE mode = 'LOW'
  RETURNING *;
```

```
 num | mode
-----+------
   1 | HIGH
(1 строка)

UPDATE 1
```

На втором терминале тоже начнем транзакцию и обновим другую строку из тех двух строк, которые были показаны выше.

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN
```

```
UPDATE modes
  SET mode = 'LOW'
  WHERE mode = 'HIGH'
  RETURNING *;
```

Изменение, произведенное в первой транзакции, вторая транзакция не видит, поскольку на уровне изоляции SERIALIZABLE каждая транзакция работает с тем снимком базы данных, которых был сделан в ее начале, т. е. непосредственно перед выполнением ее первого оператора. Поэтому обновляется только одна строка, та, в которой значение поля mode было равно «HIGH» изначально.

```
 num | mode
-----+------
   2 | LOW
(1 строка)
```

```
UPDATE 1
```

Обратите внимание, что обе команды UPDATE были выполнены, ни одна из них не ожидает завершения другой транзакции.

Посмотрим, что получилось в первой транзакции:

```
SELECT * FROM modes;
```

```
 num | mode
-----+------
   2 | HIGH
   1 | HIGH
(2 строки)
```

А во второй транзакции:

```
SELECT * FROM modes;
```

```
 num | mode
-----+------
   1 | LOW
   2 | LOW
(2 строки)
```

Заканчиваем эксперимент. Сначала завершим транзакцию на первом терминале:

```
COMMIT;
```

```
COMMIT
```

А потом на втором терминале:

```
COMMIT;
```

```
ОШИБКА: не удалось сериализовать доступ из-за зависимостей чтения/записи
 ↪  между транзакциями
ПОДРОБНОСТИ: Reason code: Canceled on identification as a pivot, during
 ↪  commit attempt.
ПОДСКАЗКА: Транзакция может завершиться успешно при следующей попытке.
```

Какое же изменение будет зафиксировано? То, которое сделала транзакция, первой выполнившая фиксацию изменений.

```
SELECT * FROM modes;
```

```
 num | mode
-----+------
   2 | HIGH
   1 | HIGH
(2 строки)
```

Таким образом, параллельное выполнение двух транзакций сериализовать не удалось. Почему? Если обратиться к определению концепции сериализации, то нужно рассуждать так. Если бы была зафиксирована и вторая транзакция, тогда в таблице modes содержались бы такие строки:

```
 num | mode
-----+------
   1 | HIGH
   2 | LOW
```

Но этот результат не соответствует результату выполнения транзакций *ни при одном* из двух возможных вариантов их упорядочения, если бы они выполнялись последовательно. Следовательно, с точки зрения концепции сериализации, эти транзакции невозможно сериализовать. Покажем это, выполнив транзакции последовательно. Предварительно необходимо пересоздать таблицу modes или с помощью команды UPDATE вернуть ее измененным строкам исходное состояние.

Теперь обе транзакции можно выполнять на одном терминале. Первый вариант их упорядочения такой:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN

UPDATE modes
  SET mode = 'HIGH'
  WHERE mode = 'LOW'
  RETURNING *;

 num | mode
-----+------
   1 | HIGH
(1 строка)

UPDATE 1

END;

COMMIT

BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN

UPDATE modes
  SET mode = 'LOW'
  WHERE mode = 'HIGH'
  RETURNING *;

 num | mode
-----+------
   2 | LOW
   1 | LOW
(2 строки)

UPDATE 2

END;

COMMIT
```

Проверим, что получилось:

```
SELECT * FROM modes;

 num | mode
-----+------
   2 | LOW
   1 | LOW
(2 строки)
```

Во втором варианте упорядочения поменяем транзакции местами. Конечно, предварительно нужно привести таблицу в исходное состояние.

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN
```

```
UPDATE modes
  SET mode = 'LOW'
  WHERE mode = 'HIGH'
  RETURNING *;
```

```
 num | mode
-----+------
   2 | LOW
(1 строка)
```

```
UPDATE 1
```

```
END;
```

```
COMMIT
```

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN
```

```
UPDATE modes
  SET mode = 'HIGH'
  WHERE mode = 'LOW'
  RETURNING *;
```

```
 num | mode
-----+------
   1 | HIGH
   2 | HIGH
(2 строки)
```

```
UPDATE 2
```

```
END;
```

```
COMMIT
```

```
SELECT * FROM modes;
```

Теперь результат отличается от того, который был получен при реализации первого варианта упорядочения транзакций.

```
 num | mode
-----+------
   1 | HIGH
   2 | HIGH
(2 строки)
```

Изменение порядка выполнения транзакций приводит к разным результатам. Однако если бы при параллельном выполнении транзакций была зафиксирована и вторая из них, то полученный результат не соответствовал бы *ни одному* из продемонстрированных возможных результатов последовательного выполнения транзакций. Таким образом, выполнить сериализацию этих транзакций невозможно. Обратите внимание, что вторая команда UPDATE в обоих случаях обновляет не одну строку, а две.

## 9.5 Пример использования транзакций

Продемонстрируем использование транзакций на примере базы данных «Авиаперевозки». Для этого создадим новое бронирование и оформим два билета с двумя перелетами в каждом. Выберем в качестве уровня изоляции READ COMMITTED.

```
BEGIN;
```

```
BEGIN;
```

Сначала добавим запись в таблицу «Бронирования», причем, значение поля total_amount назначим равным 0. После завершения ввода строк в таблицу «Перелеты» мы обновим это значение: оно станет равным сумме стоимостей всех забронированных перелетов. В качестве даты бронирования возьмем дату, которая была принята в качестве текущей в базе данных. Эту дату выдает функция now(), созданная в схеме bookings.

```
INSERT INTO bookings
  ( book_ref, book_date, total_amount )
  VALUES ( 'ABC123', bookings.now(), 0 );
```

```
INSERT 0 1
```

Оформим два билета на двух разных пассажиров.

```
INSERT INTO tickets
  ( ticket_no, book_ref, passenger_id,
    passenger_name)
  VALUES ( '9991234567890', 'ABC123', '1234 123456',
    'IVAN PETROV' );
```

```
INSERT 0 1
```

```
INSERT INTO tickets
  ( ticket_no, book_ref, passenger_id,
    passenger_name)
  VALUES ( '9991234567891', 'ABC123', '4321 654321',
    'PETR IVANOV' );
```

```
INSERT 0 1
```

Отправим обоих пассажиров по маршруту Москва — Красноярск и обратно.

```
INSERT INTO ticket_flights
  ( ticket_no, flight_id,
    fare_conditions, amount )
  VALUES ( '9991234567890', 5572, 'Business', 12500 ),
         ( '9991234567890', 13881, 'Economy', 8500 );
```

```
INSERT 0 2
```

```
INSERT INTO ticket_flights
  ( ticket_no, flight_id,
    fare_conditions, amount )
  VALUES ( '9991234567891', 5572, 'Business', 12500 ),
         ( '9991234567891', 13881, 'Economy', 8500 );
```

```
INSERT 0 2
```

Подсчитаем общую стоимость забронированных билетов и запишем ее в строку таблицы «Бронирования». Конечно, если такая транзакция выполняется в рамках прикладной программы, то возможно, что подсчет общей суммы будет выполняться в этой программе. Тогда в команде UPDATE уже не потребуется выполнять подзапрос, а будет использоваться заранее вычисленное значение. Но более надежным решением было бы использование триггера для увеличения значения поля total_amount при каждом добавлении строки в таблицу ticket_flights. Триггеры будут рассмотрены во второй части учебного пособия.

```
UPDATE bookings
  SET total_amount = (
    SELECT sum( amount )
      FROM ticket_flights
      WHERE ticket_no IN (
        SELECT ticket_no
          FROM tickets
          WHERE book_ref = 'ABC123'
      )
  )
  WHERE book_ref = 'ABC123';
```

```
UPDATE 1
```

Проверим, что получилось.

```
SELECT * FROM bookings WHERE book_ref = 'ABC123';
```

```
 book_ref |        book_date       | total_amount
----------+------------------------+--------------
 ABC123   | 2016-10-13 22:00:00+08 |     42000.00
(1 строка)
```

```
COMMIT;
```

```
COMMIT;
```

В начале главы говорилось о свойствах транзакций. Их удобно прокомментировать на примере этой транзакции, в которой участвуют три таблицы. *Атомарность* говорит о том, что либо транзакция выполняется и фиксируется полностью, либо не фиксируется ни одна из ее операций. Поэтому в случае отказа сервера баз данных в процессе выполнения транзакции и последующего восстановления состояния базы данных те операции, которые уже были выполнены, будут отменены. Таким образом, база данных будет приведена к тому *согласованному* состоянию, в котором она находилась до начала транзакции. При выборе соответствующего уровня *изоляции* эта транзакция сможет выполняться, не подвергаясь помехам со стороны других параллельных транзакций. После успешной фиксации всех выполненных изменений в базе данных пользователь может быть уверен, что они станут *долговечными* и сохранятся даже в случае сбоя в работе сервера.

## 9.6 Блокировки

Кроме поддержки уровней изоляции транзакций, PostgreSQL позволяет также создавать явные блокировки данных как на уровне отдельных строк, так и на уровне целых таблиц. Блокировки могут быть востребованы при проектировании транзакций с уровнем изоляции, как правило, READ COMMITTED, когда требуется более детальное управление параллельным выполнением транзакций. PostgreSQL предлагает много различных видов блокировок, но мы ограничимся рассмотрением только двух из них.

Команда SELECT имеет предложение FOR UPDATE, которое позволяет заблокировать отдельные строки таблицы с целью их последующего обновления. Если одна транзакция заблокировала строки с помощью этой команды, тогда параллельные транзакции не смогут заблокировать эти же строки до тех пор, пока первая транзакция не завершится, и тем самым блокировка не будет снята.

Проведем эксперимент, как и прежде, с использованием двух терминалов. Мы не будем приводить все вспомогательные команды создания и завершения транзакций, а ограничимся только командами, выполняющими полезную работу.

Итак, на первом терминале организуйте транзакцию с уровнем изоляции READ COMMITTED и выполните следующую команду:

```
SELECT * FROM aircrafts_tmp WHERE model ~ '^Air' FOR UPDATE;
```

```
 aircraft_code |        model     | range
---------------+------------------+-------
 320           | Airbus A320-200 |  5700
 321           | Airbus A321-200 |  5600
 319           | Airbus A319-100 |  6700
(3 строки)
```

На втором терминале организуйте аналогичную транзакцию и выполните точно такую же команду. Вы увидите, что ее выполнение будет приостановлено.

```
SELECT * FROM aircrafts_tmp WHERE model ~ '^Air' FOR UPDATE;
```

На первом терминале обновите одну строку, а затем завершите транзакцию:

```
UPDATE aircrafts_tmp
  SET range = 5800
  WHERE aircraft_code = '320';
```

```
UPDATE 1
```

Перейдя на второй терминал, вы увидите, что там была, наконец, выполнена выборка, которая показала уже измененные данные:

```
 aircraft_code |        model     | range
---------------+------------------+-------
 320           | Airbus A320-200 |  5800
 321           | Airbus A321-200 |  5600
 319           | Airbus A319-100 |  6700
(3 строки)
```

Завершите и вторую транзакцию.

Аналогичным образом можно организовать блокировки на уровне таблиц. Также на первом терминале организуйте транзакцию с уровнем изоляции READ COMMITTED и выполните команду блокировки всей таблицы в самом строгом режиме, в котором другим транзакциям доступ к этой таблице запрещен полностью:

```
LOCK TABLE aircrafts_tmp IN ACCESS EXCLUSIVE MODE;
```

```
LOCK TABLE
```

На втором терминале выполните совершенно «безобидную» команду:

```
SELECT * FROM aircrafts_tmp WHERE model ~ '^Air';
```

Вы увидите, что выполнение команды SELECT на втором терминале будет задержано. Прервите транзакцию на первом терминале командой ROLLBACK. Вы увидите, что на втором терминале команда будет успешно выполнена.

Более подробно ознакомиться с различными видами блокировок уровня строки и уровня таблицы можно с помощью документации (раздел 13.3 «Явные блокировки»).

## Контрольные вопросы и задания

1. По умолчанию каждая SQL-команда, выполняемая в среде psql, образует отдельную транзакцию с уровнем изоляции READ COMMITTED. Поэтому в тех экспериментах, когда одна из транзакций состоит только из единственной SQL-команды, можно не выполнять команды BEGIN и END. Конечно, если каждая из параллельных транзакций состоит из единственной SQL-команды, то хотя бы для одной из транзакций придется все же выполнить и команду BEGIN, иначе эксперимент не получится.

   В тексте главы были приведены примеры транзакций, в которых рассматривались команды SELECT … FOR UPDATE и LOCK TABLE. Попробуйте повторить эти эксперименты с учетом описанного поведения PostgreSQL.

2. Транзакции, работающие на уровне изоляции READ COMMITTED, видят только свои собственные обновления и обновления, *зафиксированные* параллельными транзакциями. При этом нужно учитывать, что иногда могут возникать ситуации, которые на первый взгляд кажутся парадоксальными, но на самом деле все происходит в строгом соответствии с этим принципом.

   Давайте воспользуемся таблицей «Самолеты» (aircrafts) или ее копией. Предположим, что мы решили удалить из таблицы те модели, дальность полета которых менее 2000 км. В таблице представлена одна такая модель — Cessna 208 Caravan, имеющая дальность полета 1200 км. Для выполнения удаления мы организовали транзакцию. Однако параллельная транзакция, которая, причем, началась раньше, успела обновить таблицу таким образом, что дальность полета самолета Cessna 208 Caravan стала составлять 2100 км, а вот для самолета Bombardier CRJ-200 она, напротив, уменьшилась до 1900 км. Таким образом, в результате выполнения операций обновления в таблице по-прежнему присутствует строка, удовлетворяющая первоначальному условию, т. е. значение атрибута range у которой меньше 2000.

Наша задача: проверить, будет ли в результате выполнения двух транзакций удалена какая-либо строка из таблицы.

На первом терминале начнем транзакцию, при этом уровень изоляции READ COMMITTED в команде указывать не будем, т. к. он принят по умолчанию:

```
BEGIN;
```

```
BEGIN
```

```
SELECT * FROM aircrafts_tmp WHERE range < 2000;
```

```
 aircraft_code |        model       | range
---------------+--------------------+-------
 CN1           | Cessna 208 Caravan |  1200
(1 строка)
```

```
UPDATE aircrafts_tmp
  SET range = 2100
  WHERE aircraft_code = 'CN1';
```

```
UPDATE 1
```

```
UPDATE aircrafts_tmp
  SET range = 1900
  WHERE aircraft_code = 'CR2';
```

```
UPDATE 1
```

На втором терминале начнем вторую транзакцию, которая и будет пытаться удалить строки, у которых значение атрибута range меньше 2000.

```
BEGIN;
```

```
BEGIN
```

```
SELECT * FROM aircrafts_tmp WHERE range < 2000;
```

```
 aircraft_code |        model       | range
---------------+--------------------+-------
 CN1           | Cessna 208 Caravan |  1200
(1 строка)
```

```
DELETE FROM aircrafts_tmp WHERE range < 2000;
```

Введя команду DELETE, мы видим, что она не завершается, а ожидает, когда со строки, подлежащей удалению, будет снята блокировка. Блокировка, установленная командой UPDATE в первой транзакции, снимается только при завершении транзакции, а завершение может иметь два исхода: фиксацию изменений с помощью команды COMMIT (или END) или отмену изменений с помощью команды ROLLBACK.

Давайте зафиксируем изменения, выполненные первой транзакцией. На первом терминале сделаем так:

```
COMMIT;
```

```
COMMIT
```

Тогда на втором терминале мы получим такой результат от команды DELETE:

```
DELETE 0
```

Чем объясняется такой результат? Он кажется нелогичным: ведь команда SELECT, выполненная в этой же второй транзакции, показывала наличие строки, удовлетворяющей условию удаления.

Объяснение таково: поскольку вторая транзакция пока еще не видит изменений, произведенных в первой транзакции, то команда DELETE выбирает для удаления строку, описывающую модель Cessna 208 Caravan, однако эта строка была заблокирована в первой транзакции командой UPDATE. Эта команда изменила значение атрибута range в этой строке. При завершении первой транзакции блокировка с этой строки снимается (со второй строки — тоже), и команда DELETE во второй транзакции получает возможность заблокировать эту строку. При этом команда DELETE данную строку *перечитывает* и вновь вычисляет условие WHERE применительно к ней. Однако теперь условие WHERE для данной строки уже не выполняется, следовательно, эту строку удалять нельзя. Конечно, в таблице есть теперь другая строка, для самолета Bombardier CRJ-200, удовлетворяющая условию удаления, однако повторный поиск строк, удовлетворяющих условию WHERE в команде DELETE, не производится. В результате не удаляется ни одна строка. Таким образом, к сожалению, имеет место нарушение согласованности, которое можно объяснить деталями реализации СУБД.

Завершим вторую транзакцию:

```
END;
```

```
COMMIT
```

Вот что получилось в результате:

```
SELECT * FROM aircrafts_tmp;
```

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 773           | Boeing 777-300       | 11100
 763           | Boeing 767-300       |  7900
 SU9           | Sukhoi SuperJet-100  |  3000
 320           | Airbus A320-200      |  5700
 321           | Airbus A321-200      |  5600
 319           | Airbus A319-100      |  6700
 733           | Boeing 737-300       |  4200
 CN1           | Cessna 208 Caravan   |  2100
 CR2           | Bombardier CRJ-200   |  1900
(9 строк)
```

Задание: модифицируйте сценарий выполнения транзакций, а именно: в первой транзакции вместо фиксации изменений выполните их отмену с помощью команды ROLLBACK и посмотрите, будет ли удалена строка и какая конкретно.

3.* Когда говорят о таком феномене, как потерянное обновление, то зачастую в качестве примера приводится операция UPDATE, в которой значение какого-то атрибута изменяется с применением одного из действий арифметики. Например:

```
UPDATE aircrafts_tmp SET range = range + 200
WHERE aircraft_code = 'CR2';
```

При выполнении двух и более подобных обновлений в рамках параллельных транзакций, использующих, например, уровень изоляции READ COMMITTED, будут учтены все такие изменения (что и было показано в тексте главы). Очевидно, что потерянного обновления не происходит.

Предположим, что в одной транзакции будет просто присваиваться новое значение, например, так:

```
UPDATE aircrafts_tmp SET range = 2100 WHERE aircraft_code = 'CR2';
```

А в параллельной транзакции будет выполняться аналогичная команда, например:

```
UPDATE aircrafts_tmp SET range = 2500 WHERE aircraft_code = 'CR2';
```

Очевидно, что сохранится только одно из значений атрибута range. Можно ли говорить, что в такой ситуации имеет место потерянное обновление? Если оно имеет место, то что можно предпринять для его недопущения? Обоснуйте ваш ответ.

Для получения дополнительной информации можно обратиться к фундаментальному труду К. Дж. Дейта, а также к полному руководству по SQL Дж. Гроффа, П. Вайнберга и Э. Оппеля. Библиографические описания этих книг приведены в списке рекомендуемой литературы.

4. На уровне изоляции транзакций READ COMMITTED имеет место такой феномен, как чтение фантомных строк. Такие строки могут появляться в выборке как в результате добавления новых строк параллельной транзакцией, так и вследствие изменения ею значений атрибутов, участвующих в формировании условия выборки. Рассмотрим пример, иллюстрирующий вторую из указанных причин.

   На первом терминале организуем транзакцию с уровнем изоляции READ COMMITTED:

```
BEGIN;
```

```
BEGIN
```

```
SELECT * FROM aircrafts_tmp WHERE range > 6000;
```

```
 aircraft_code |       model     | range
---------------+-----------------+-------
 773           | Boeing 777-300  | 11100
 763           | Boeing 767-300  |  7900
 319           | Airbus A319-100 |  6700
(3 строки)
```

   На втором терминале организуем транзакцию и обновим одну из строк таблицы таким образом, чтобы эта строка стала удовлетворять условию отбора строк, заданному в первой транзакции.

```
BEGIN;
```

```
BEGIN
```

```
UPDATE aircrafts_tmp
  SET range = 6100
  WHERE aircraft_code = '320';
```

```
UPDATE 1
```

Сразу завершим вторую транзакцию, чтобы первая транзакция увидела эти изменения.

```
END;
```

```
COMMIT
```

На первом терминале повторим ту же самую выборку:

```
SELECT * FROM aircrafts_tmp WHERE range > 6000;
```

```
 aircraft_code |        model     | range
---------------+------------------+-------
 773           | Boeing 777-300   | 11100
 763           | Boeing 767-300   |  7900
 319           | Airbus A319-100  |  6700
 320           | Airbus A320-200  |  6100
(4 строки)
```

Транзакция еще не завершилась, но она уже увидела новую строку, обновленную зафиксированной параллельной транзакцией. Теперь эта строка стала соответствовать условию выборки. Таким образом, не изменяя критерий выборки, мы получили другое множество строк.

Завершим теперь и первую транзакцию:

```
END;
```

```
COMMIT
```

Задание: модифицируйте этот эксперимент: вместо операции UPDATE используйте операцию INSERT.

5. В тексте главы была рассмотрена команда SELECT … FOR UPDATE, выполняющая блокировку на уровне отдельных строк. Организуйте две параллельные транзакции с уровнем изоляции READ COMMITTED и выполните с ними ряд экспериментов. В первой транзакции заблокируйте некоторое множество строк, отбираемых с помощью условия WHERE. А во второй транзакции изменяйте условие выборки таким образом, чтобы выбираемое множество строк:

   – являлось подмножеством множества строк, выбираемых в первой транзакции;

   – являлось надмножеством множества строк, выбираемых в первой транзакции;

   – пересекалось с множеством строк, выбираемых в первой транзакции;

   – не пересекалось с множеством строк, выбираемых в первой транзакции.

Наблюдайте за поведением команд выборки в каждой транзакции. Попробуйте обобщить ваши наблюдения.

6. Самостоятельно ознакомьтесь с предложением FOR SHARE команды SELECT и выполните необходимые эксперименты. Используйте документацию: раздел 13.3.2 «Блокировки на уровне строк» и описание команды SELECT.

7. В тексте главы для иллюстрации изучаемых концепций мы создавали только две параллельные транзакции. Попробуйте воспроизвести представленные эксперименты, создав три или даже четыре параллельные транзакции.

8.* В тексте главы была рассмотрена транзакция для выполнения бронирования билетов. Для нее был выбран уровень изоляции READ COMMITTED.

   Как вы думаете, если одновременно будут производиться несколько операций бронирования, то, может быть, имеет смысл «ужесточить» уровень изоляции до SERIALIZABLE? Или нет необходимости это делать? Обдумайте и вариант с использованием явных блокировок. Обоснуйте ваш ответ.

9.* В разделе документации 13.2.3 «Уровень изоляции Serializable» сказано, что если поиск в таблице осуществляется последовательно, без использования индекса, тогда на всю таблицу накладывается так называемая предикатная блокировка. Такой подход приводит к увеличению числа сбоев сериализации. В качестве контрмеры можно попытаться использовать индексы. Конечно, если таблица совсем небольшая, то может и не получиться заставить PostgreSQL использовать поиск по индексу. Тем не менее, давайте выполним следующий эксперимент.

   Для его проведения создадим специальную таблицу, в которой будет всего два столбца: один — числовой, а второй — текстовый. Значения во втором столбце будут иметь вид: LOW1, LOW2, ..., HIGH1, HIGH2, ... Назовем эту таблицу — modes. Добавим в нее такое число строк, которое сделает очень вероятным использование индекса при выполнении операций обновления строк и, соответственно, отсутствие предикатной блокировки всей таблицы. О том, как узнать, используется ли индекс при выполнении тех или иных операций, написано в главе 10.

```
CREATE TABLE modes AS
  SELECT num::integer, 'LOW' || num::text AS mode
    FROM generate_series( 1, 100000 ) AS gen_ser( num )
  UNION ALL
  SELECT num::integer, 'HIGH' || ( num - 100000 )::text AS mode
    FROM generate_series( 100001, 200000 ) AS gen_ser( num );
```

```
SELECT 200000
```

   Проиндексируем таблицу по числовому столбцу.

```
CREATE INDEX modes_ind ON modes ( num );
```

```
CREATE INDEX
```

   Из всего множества строк нас будут интересовать только две:

```
SELECT * FROM modes WHERE mode IN ( 'LOW1', 'HIGH1' );
```

```
  num   | mode
--------+-------
      1 | LOW1
 100001 | HIGH1
(2 строки)
```

На первом терминале начнем транзакцию и обновим одну строку из тех двух строк, которые были показаны в предыдущем запросе.

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN
```

```
UPDATE modes SET mode = 'HIGH1' WHERE num = 1;
```

```
UPDATE 1
```

На втором терминале тоже начнем транзакцию и обновим другую строку из тех двух строк, которые были показаны выше.

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN
```

```
UPDATE modes SET mode = 'LOW1' WHERE num = 100001;
```

```
UPDATE 1
```

Обратите внимание, что обе команды UPDATE были выполнены, ни одна из них не ожидает завершения другой транзакции.

Попробуем завершить транзакции. Сначала — на первом терминале:

```
COMMIT;
```

```
COMMIT
```

А потом на втором терминале:

```
COMMIT;
```

```
COMMIT
```

Посмотрим, что получилось.

```
SELECT * FROM modes WHERE mode IN ( 'LOW1', 'HIGH1' );
```

```
  num   | mode
--------+-------
      1 | HIGH1
 100001 | LOW1
(2 строки)
```

Теперь система смогла сериализовать параллельные транзакции и зафиксировать их обе. Как вы думаете, почему это удалось? Обосновывая ваш ответ, примите во внимание тот результат, который был бы получен при последовательном выполнении транзакций.

10.* В тексте главы был рассмотрен пример транзакции над таблицами базы данных «Авиаперевозки». Давайте теперь создадим две параллельные транзакции и выполним их с уровнем изоляции SERIALIZABLE. Отправим также двоих пассажиров теми же самыми рейсами, что и ранее, но операции распределим между двумя транзакциями. Отличие заключается в том, что в начале транзакции будут выполняться выборки из таблицы ticket_flights. Для упрощения ситуации не будем предварительно проверять наличие свободных мест, т. к. сейчас для нас важно не это. Итак, первая транзакция:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN
```

```
SELECT * FROM ticket_flights WHERE flight_id = 13881;
```

```
   ticket_no   | flight_id | fare_conditions | amount
---------------+-----------+-----------------+----------
 0005433848165 |     13881 | Business        | 99800.00
 ...
 0005433848007 |     13881 | Economy         | 33300.00
(82 строки)
```

```
INSERT INTO bookings ( book_ref, book_date, total_amount )
  VALUES ( 'ABC123', bookings.now(), 0 );
```

```
INSERT 0 1
```

```
INSERT INTO tickets
  ( ticket_no, book_ref, passenger_id, passenger_name )
  VALUES
  ( '9991234567890', 'ABC123', '1234 123456',
    'IVAN PETROV' );
```

```
INSERT 0 1
```

```
INSERT INTO ticket_flights
  ( ticket_no, flight_id, fare_conditions, amount )
  VALUES
  ( '9991234567890', 13881, 'Business', 12500 );
```

```
INSERT 0 1
```

```
UPDATE bookings
  SET total_amount = 12500
  WHERE book_ref = 'ABC123';
```

```
UPDATE 1
```

```
COMMIT;
```

```
COMMIT
```

Вторая транзакция:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN
```

```
SELECT * FROM ticket_flights WHERE flight_id = 5572;
```

```
  ticket_no    | flight_id | fare_conditions | amount
---------------+-----------+-----------------+----------
 0005433847924 |      5572 | Business        | 99800.00
 ...
 0005433847890 |      5572 | Economy         | 33300.00
(100 строк)
```

```
INSERT INTO bookings
  ( book_ref, book_date, total_amount )
  VALUES
  ( 'ABC456', bookings.now(), 0 );
```

```
INSERT 0 1
```

```
INSERT INTO tickets
  ( ticket_no, book_ref, passenger_id, passenger_name )
  VALUES
  ( '9991234567891', 'ABC456', '4321 654321',
    'PETR IVANOV' );
```

```
INSERT 0 1
```

```
INSERT INTO ticket_flights
  ( ticket_no, flight_id, fare_conditions, amount )
  VALUES
  ( '9991234567891', 5572, 'Business', 12500 );
```

```
INSERT 0 1
```

```
UPDATE bookings
  SET total_amount = 12500
  WHERE book_ref = 'ABC456';
```

```
UPDATE 1
```

```
COMMIT;
```

```
ОШИБКА: не удалось сериализовать доступ из-за зависимостей
 ↪  чтения/записи между транзакциями
ПОДРОБНОСТИ: Reason code: Canceled on identification as a pivot,
 ↪  during commit attempt.
ПОДСКАЗКА: Транзакция может завершиться успешно при следующей
 ↪  попытке.
```

**Задание 1.** Попытайтесь объяснить, почему транзакции не удалось сериализовать. Что можно сделать, чтобы удалось зафиксировать обе транзакции? Одно из возможных решений — понизить уровень изоляции. Другим решением может быть создание индекса по столбцу flight_id для таблицы ticket_flights. Почему создание индекса может помочь? Обратитесь за разъяснениями к разделу документации 13.2.3 «Уровень изоляции Serializable».

**Задание 2.** В первой транзакции условие в команде SELECT такое: ...WHERE flight_id = 13881. В команде вставки в таблицу ticket_flights значение поля

flight_id также равно 13881. Во второй транзакции в этих же командах используется значение 5572. Поменяйте местами значения в командах SELECT и повторите эксперименты, выполнив транзакции параллельно с уровнем изоляции SERIALIZABLE. Почему сейчас наличие индекса не помогает зафиксировать обе транзакции? Вспомните, что аномалия сериализации — это ситуация, когда параллельное выполнение транзакций приводит к результату, невозможному ни при каком из вариантов упорядочения этих же транзакций при их последовательном выполнении.

# 10 Повышение производительности

Заставить PostgreSQL работать быстро — эта задача может возникнуть с ростом объема данных. Мы покажем лишь самые простые методы ее решения.

## 10.1 Основные понятия

Для понимания материала этой главы необходимо сначала познакомиться с такими важными понятиями, как метод доступа и способ соединения наборов строк.

**Метод доступа** характеризует тот способ, который используется для просмотра таблиц и извлечения только тех строк, которые соответствуют критерию отбора. Существуют различные методы доступа: последовательный просмотр (sequential scan), при котором индекс не используется, и группа методов, основанных на использовании индекса. К ней относятся: просмотр по индексу (index scan), просмотр исключительно на основе индекса (index only scan) и просмотр на основе битовой карты (bitmap scan).

Поскольку и таблицы, и индексы хранятся на диске, то для работы с ними эти объекты считываются в память, в которой они представлены разбитыми на отдельные фрагменты, называемые страницами. Эти страницы имеют специальную структуру. Размер страниц по умолчанию составляет 8 килобайт.

При выполнении **последовательного просмотра** (sequential scan) обращения к индексам не происходит, а строки извлекаются из табличных страниц в соответствии с критерием отбора. В том случае, когда в запросе нет предложения WHERE, тогда извлекаются все строки таблицы. Данный метод применяется, когда требуется выбрать все строки таблицы или значительную их часть, т. е. когда так называемая *селективность* выборки низка. В таком случае обращение к индексу не ускорит процесс просмотра, а возможно даже и замедлит.

**Просмотр на основе индекса** (index scan) предполагает обращение к индексу, созданному для данной таблицы. Поскольку в индексе для каждого ключевого значения содержатся уникальные идентификаторы строк в таблицах, то после отыскания в индексе нужного ключа производится обращение к соответствующей странице таблицы и извлечение искомой строки по ее идентификатору. При этом нужно учитывать, что хотя записи в индексе упорядочены, но обращения к страницам таблицы происходят хаотически, поскольку строки в таблицах не упорядочены. В таком случае при низкой селективности выборки, т. е. когда из таблицы отбирается значительное число строк, использование индексного поиска может не только не давать ускорения работы, но даже и снижать производительность.

**Просмотр исключительно на основе индекса** (index only scan), как следует из названия метода, не должен, казалось бы, требовать обращения к строкам таблицы, поскольку все данные, которые нужно получить с помощью запроса, в этом случае присутствуют в индексе. Однако в индексе нет информации о *видимости* строк транзакциям — нельзя быть уверенным, что данные, полученные из индекса, видны текущей транзакции. Поэтому сначала выполняется обращение к карте видимости (visibility

map), которая существует для каждой таблицы. В ней одним битом отмечены страницы, на которых содержатся только те версии строк, которые видны всем без исключения транзакциям. Если полученная из индекса версия строки находится на такой странице, значит, эта строка видна текущей транзакции и обращаться к самой таблице не требуется. Поскольку размер карты видимости очень мал, то в результате сокращается объем операций ввода/вывода. Если же строка находится на странице, не отмеченной в карте видимости, тогда происходит обращение и к таблице; в результате никакого выигрыша по быстродействию в сравнении с обычным индексным поиском не достигается. Просмотр исключительно на основе индекса особенно эффективен, когда выбираемые данные изменяются редко. Он может применяться, когда в предложении SELECT указаны только имена столбцов, по которым создан индекс.

**Просмотр на основе битовой карты** (bitmap scan) является модификацией просмотра на основе индекса. Данный метод позволяет оптимизировать индексный поиск за счет того, что сначала производится поиск в индексе для всех искомых строк и формирование так называемой битовой карты, в которой указывается, в каких страницах таблицы эти строки содержатся. После того как битовая карта сформирована, выполняется извлечение строк из страниц таблицы, но при этом обращение к каждой странице производится *только один* раз.

Другим важным понятием является **способ соединения наборов строк** (join). Набор строк может быть получен из таблицы с помощью одного их методов доступа, описанных выше. Набор строк может быть получен не только из одной таблицы, а может быть результатом соединения других наборов. Важно различать способ соединения таблиц (JOIN) и способ соединения наборов строк. Первое понятие относится к языку SQL и является высокоуровневым, логическим, оно не касается вопросов реализации. А второе относится именно к реализации, это — механизм непосредственного выполнения соединения наборов строк. Принципиально важным является то, что за один раз соединяются только два набора строк.

Существует три способа соединения: вложенный цикл (nested loop), хеширование (hash join) и слияние (merge join). Они имеют свои особенности, которые PostgreSQL учитывает при выполнении конкретных запросов.

Суть способа **«вложенный цикл»** в том, что перебираются строки из «внешнего» набора и для каждой из них выполняется поиск соответствующих строк во «внутреннем» наборе. Если соответствующие строки найдены, то выполняется их соединение со строкой из «внешнего» набора. При этом способы выбора строк из обоих наборов могут быть различными. Метод поддерживает соединения как на основе равенства значений атрибутов (эквисоединения), так и любые другие виды условий. Поскольку он не требует подготовительных действий, то способен быстро приступить к непосредственной выдаче результата. Метод эффективен для небольших выборок.

При **соединении хешированием** строки одного набора помещаются в хеш-таблицу, содержащуюся в памяти, а строки из второго набора перебираются, и для каждой из них проверяется наличие соответствующих строк в хеш-таблице. Ключом хеш-таблицы является тот столбец, по которому выполняется соединение наборов строк. Как правило, число строк в том наборе, на основе которого строится хеш-таблица, меньше, чем во втором наборе. Это позволяет уменьшить ее размер и ускорить процесс обращения к ней. Данный метод работает только при выполнении эквисоедине-

ний, поскольку для хеш-таблицы имеет смысл только проверка на равенство проверяемого значения одному из ее ключей. Метод эффективен для больших выборок.

**Соединение методом слияния** производится аналогично сортировке слиянием. В этом случае оба набора строк должны быть предварительно отсортированы по тем столбцам, по которым производится соединение. Затем параллельно читаются строки из обоих наборов и сравниваются значения столбцов, по которым производится соединение. При совпадении значений формируется результирующая строка. Этот процесс продолжается до исчерпания строк в обоих наборах. Этот метод, как и метод соединения хешированием, работает только при выполнении эквисоединений. Он пригоден для работы с большими наборами строк.

## 10.2 Методы просмотра таблиц

Теперь мы можем перейти к рассмотрению планов выполнения запросов.

Прежде чем приступить к непосредственному выполнению каждого запроса, PostgreSQL формирует *план* его выполнения. Чтобы достичь хорошей производительности, этот план должен учитывать свойства данных. Планированием занимается специальная подсистема — планировщик (planner). Просмотреть план выполнения запроса можно с помощью команды EXPLAIN. Для детального понимания планов выполнения сложных запросов требуется опыт. Мы изложим лишь основные приемы работы с этой командой.

Структура плана запроса представляет собой дерево, состоящее из так называемых *узлов плана* (plan nodes). Узлы на нижних уровнях дерева отвечают за просмотр и выдачу строк таблиц, которые осуществляются с помощью методов доступа, описанных выше. Если конкретный запрос требует выполнения операций агрегирования, соединения таблиц, сортировки, то над узлами выборки строк будут располагаться дополнительные узлы дерева плана. Например, для соединения наборов строк будут использоваться способы, которые мы только что рассмотрели. Для каждого узла дерева плана команда EXPLAIN выводит по одной строке, при этом выводятся также оценки стоимости выполнения операций на каждом узле, которые делает планировщик. В случае необходимости для конкретных узлов могут выводиться дополнительные строки. Самая первая строка плана содержит общую оценку стоимости выполнения данного запроса.

Запустите утилиту psql и введите простой запрос:

```
EXPLAIN SELECT * FROM aircrafts;
```

В ответ получим план выполнения запроса:

```
                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on aircrafts (cost=0.00..1.09 rows=9 width=52)
(1 строка)
```

Поскольку в этом запросе нет предложения WHERE, он должен просмотреть все строки таблицы, поэтому планировщик выбирает **последовательный просмотр** (sequential scan). В скобках приведены важные параметры плана.

Первое число означает оценку ресурсов, требуемых для того, чтобы приступить к выводу данных. В нашем примере эта оценка равна нулю, поскольку никакие дополнительные операции с выбранными строками не предполагаются, и PostgreSQL может сразу же выводить прочитанные строки.

Второе число — это оценка общей стоимости выполнения запроса. Формируя эту оценку, планировщик исходит из предположения, что данный узел плана запроса выполняется до конца, т. е. извлекаются все имеющиеся строки таблицы. Однако в ряде случаев на практике это может оказаться и не так, если узел-родитель прекратит свою работу досрочно, например, в случае использования в запросе SELECT предложения LIMIT, которое ограничивает выборку записей из таблицы конкретным их числом. Обе оценки стоимости выполнения выражаются в неких *условных единицах*, которые вычисляются на основе ряда параметров сервера баз данных. При этом не важно, в каких конкретно единицах производится измерение стоимости: важны соотношения стоимостей. Для каждого запроса планировщик формирует несколько планов. При сравнении различных вариантов плана, как правило, для выполнения выбирается тот, который имеет наименьшую общую стоимость выполнения запроса. Однако при работе с курсорами этот принцип можно изменить с помощью специального параметра планировщика cursor_tuple_fraction (курсоры в учебном пособии не рассматриваются).

Далее в выводе идет общее число строк, которые должны быть извлечены (возвращены) на данном узле плана, также при условии выполнения этого узла до полного завершения. В нашем примере число строк равно 9. Это число является оценкой, которую планировщик получает на основе статистики, накапливаемой в специальных системных таблицах.

Последним параметром узла плана идет оценка среднего размера строк, которые выводятся на данном узле плана запроса. В нашем примере размер (ширина) строки данных оценивается в 52 байта.

В том случае, когда нас не интересуют численные оценки, можно воспользоваться параметром COSTS OFF:

```
EXPLAIN ( COSTS OFF ) SELECT * FROM aircrafts;
```

```
      QUERY PLAN
-----------------------
 Seq Scan on aircrafts
(1 строка)
```

Сформируем запрос с предложением WHERE:

```
EXPLAIN SELECT * FROM aircrafts WHERE model ~ 'Air';
```

```
                      QUERY PLAN
-------------------------------------------------------------
 Seq Scan on aircrafts (cost=0.00..1.11 rows=1 width=52)
   Filter: (model ~ 'Air'::text)
(2 строки)
```

Поскольку наложено дополнительное условие на строки, выбираемые из таблицы, то ниже узла плана, отвечающего за их последовательную выборку, добавляется еще один узел, описывающий критерий отбора строк.

```
Filter: (model ~ 'Air'::text)
```

Поскольку наложено условие отбора строк, то оценка их числа изменилась с 9 на 1. В данном случае планировщик неточно оценил число выбираемых строк — фактически их будет три.

Обратите внимание, что по своей форме вывод команды EXPLAIN также является выборкой, поэтому в конце выборки, как обычно, выводится информация о числе строк в ней, т. е. в дереве плана. Это не число строк, которые будут выбраны из таблицы. В данном случае это

```
(2 строки)
```

Теперь усложним запрос, добавив в него сортировку данных:

```
EXPLAIN SELECT * FROM aircrafts ORDER BY aircraft_code;
```

```
                          QUERY PLAN
---------------------------------------------------------------
Sort (cost=1.23..1.26 rows=9 width=52)
  Sort Key: aircraft_code
  -> Seq Scan on aircrafts (cost=0.00..1.09 rows=9 width=52)
(3 строки)
```

Дополнительный узел обозначен на плане символами «−>».

Хотя по столбцу aircraft_code создан индекс (для поддержки первичного ключа), планировщик предпочел не использовать этот индекс, а прибегнуть к последовательному сканированию (Seq Scan) таблицы, о чем говорит нам нижний узел плана. На верхнем узле выполняется сортировка выбранных строк. Поскольку для выполнения сортировки требуется время, отличное от нуля, то этот факт и отражен в первой числовой оценке — 1,23. Это оценка времени, которое потребуется для того, чтобы приступить к выводу отсортированных строк. Но времени непосредственно на саму сортировку потребуется меньше: ведь в оценку 1,23 входит и оценка стоимости получения выборки — 1,09.

Когда таблица очень маленькая, то обращение к индексу не даст выигрыша в скорости, а лишь добавит к операциям чтения страниц, в которых хранятся строки таблиц, еще и операции чтения страниц с записями индекса.

Обратимся к таблице «Бронирования» (bookings) для иллюстрации **сканирования по индексу**.

```
EXPLAIN SELECT * FROM bookings ORDER BY book_ref;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Index Scan using bookings_pkey on bookings (cost=0.42..8511.24
 ↪  rows=262788 width=21)
(1 строка)
```

Поскольку выводимые строки плана в утилите psql могут быть очень длинными, мы будем вносить небольшие изменения в форматирование вывода при переносе плана в текст пособия.

Обратите внимание, что первая оценка стоимости в плане — не нулевая. Это объясняется тем, что, хотя индекс уже упорядочен, и дополнительная сортировка не требуется, но для того, чтобы найти в индексе первую строку в соответствии с требуемым порядком, тоже нужно некоторое время.

Если к сортировке добавить еще и условие отбора строк, то это отразится в дополнительной строке верхнего (и единственного) узла плана.

```
EXPLAIN
  SELECT * FROM bookings
    WHERE book_ref > '0000FF' AND book_ref < '000FFF'
    ORDER BY book_ref;
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Index Scan using bookings_pkey on bookings (cost=0.42..9.50 rows=54
 ↪  width=21)
   Index Cond: ((book_ref > '0000FF'::bpchar) AND (book_ref <
 ↪  '000FFF'::bpchar))
(2 строки)
```

Обратите внимание, что поскольку столбец, по которому производится отбор строк, является индексируемым, то их отбор реализуется не через Filter, а через Index Cond.

Теперь проиллюстрируем метод **сканирования на основе битовой карты** на примере таблицы «Места» (seats).

```
EXPLAIN SELECT * FROM seats WHERE aircraft_code = 'SU9';
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Bitmap Heap Scan on seats (cost=5.03..14.24 rows=97 width=15)
   Recheck Cond: (aircraft_code = 'SU9'::bpchar)
   -> Bitmap Index Scan on seats_pkey (cost=0.00..5.00 rows=97 width=0)
        Index Cond: (aircraft_code = 'SU9'::bpchar)
(4 строки)
```

В этом плане в нижнем узле строится битовая карта, а в верхнем узле с помощью этой карты сканируются страницы таблицы seats. Здесь также для отбора строк в соответствии с предложением WHERE используется индекс — Index Cond. Обратите внимание, что значение параметра width при создании битовой карты равно нулю, поскольку сами строки на этом этапе еще не выбираются.

Если нам будет нужно выбрать только номера бронирований в каком-то диапазоне, то обращения к таблице не потребуется: достаточно **сканирования исключительно по индексу**.

```
EXPLAIN
  SELECT book_ref
    FROM bookings
    WHERE book_ref < '000FFF'
    ORDER BY book_ref;
```

```
                          QUERY PLAN
-------------------------------------------------------------
 Index Only Scan using bookings_pkey on bookings (cost=0.42..9.42 rows=57
 ↪  width=7)
   Index Cond: (book_ref < '000FFF'::bpchar)
(2 строки)
```

В этом плане только один узел — Index Only Scan. Здесь также первая оценка стоимости не нулевая, т. к. отыскание в индексе наименьшего значения требует некоторого времени.

Посмотрим, как отражаются в планах выполнения запросов различные **агрегатные функции**. Начнем с простого подсчета строк.

```
EXPLAIN
  SELECT count( * )
    FROM seats
    WHERE aircraft_code = 'SU9';
```

```
                          QUERY PLAN
-------------------------------------------------------------
 Aggregate (cost=14.48..14.49 rows=1 width=8)
   -> Bitmap Heap Scan on seats (cost=5.03..14.24 rows=97 width=0)
        Recheck Cond: (aircraft_code = 'SU9'::bpchar)
        -> Bitmap Index Scan on seats_pkey (cost=0.00..5.00 rows=97
 ↪  width=0)
             Index Cond: (aircraft_code = 'SU9'::bpchar)
(5 строк)
```

В верхнем узле плана выполняется агрегирование — Aggregate. А в нижних узлах подготавливаются строки с помощью сканирования на основе формирования битовой карты.

Но возникает вопрос: зачем вообще выполняется обращение к страницам таблицы (Bitmap Heap Scan), если никакие значения атрибутов не выбираются, а подсчитывается лишь число этих строк? Казалось бы, достаточно использования только индекса. Но это нужно для того, чтобы проверить видимость версий строк: ведь разные транзакции могут видеть разные версии строк, поэтому при подсчете их числа нужно учитывать, какой транзакции они видны. Обратите еще внимание на тот факт, что собственно стадия агрегирования «стоит» не очень дорого. Ее можно приблизительно оценить как 0,24 (отняв от оценки 14,48 в узле Aggregate оценку 14,24 в узле Bitmap Heap Scan).

А в этом примере агрегирование связано уже с вычислениями на основе значений конкретного столбца, а не просто с подсчетом строк.

```
EXPLAIN SELECT avg( total_amount ) FROM bookings;
```

```
                          QUERY PLAN
-------------------------------------------------------------
 Aggregate (cost=4958.85..4958.86 rows=1 width=32)
   -> Seq Scan on bookings (cost=0.00..4301.88 rows=262788 width=6)
(2 строки)
```

## 10.3 Методы формирования соединений наборов строк

Теперь обратимся к методам, которые используются для формирования соединений наборов строк. Начнем с метода **вложенного цикла** (nested loop). Для получения списка мест в салонах самолетов Airbus с указанием класса обслуживания сформируем запрос, в котором соединяются две таблицы: «Места» (seats) и «Самолеты» (aircrafts).

```
EXPLAIN
  SELECT a.aircraft_code, a.model,
         s.seat_no, s.fare_conditions
    FROM seats s
    JOIN aircrafts a ON s.aircraft_code = a.aircraft_code
    WHERE a.model ~ '^Air'
    ORDER BY s.seat_no;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Sort (cost=23.28..23.65 rows=149 width=59)
    Sort Key: s.seat_no
    -> Nested Loop (cost=5.43..17.90 rows=149 width=59)
         -> Seq Scan on aircrafts a (cost=0.00..1.11 rows=1 width=48)
              Filter: (model ~ '^Air'::text)
         -> Bitmap Heap Scan on seats s (cost=5.43..15.29 rows=149
 ↪  width=15)
              Recheck Cond: (aircraft_code = a.aircraft_code)
              -> Bitmap Index Scan on seats_pkey (cost=0.00..5.39
 ↪  rows=149 width=0)
                   Index Cond: (aircraft_code = a.aircraft_code)
(9 строк)
```

Узел Nested Loop, в котором выполняется соединение, имеет два дочерних узла: внешний — Seq Scan и внутренний — Bitmap Heap Scan. Во внешнем узле последовательно сканируется таблица aircrafts с целью отбора строк согласно условию Filter: (model ~'^Air'::text). Для каждой из отобранных строк во внутреннем дочернем узле (Bitmap Heap Scan) выполняется поиск в таблице seats по индексу с использованием битовой карты. Она формируется в узле Bitmap Index Scan с учетом условия Index Cond: (aircraft_code = a.aircraft_code), т. е. для текущего значения атрибута aircraft_code, по которому выполняется соединение. На верхнем уровне плана сформированные строки сортируются по ключу (Sort Key: s.seat_no).

Следующий метод соединения наборов строк — **соединение хешированием** (hash join). Получим список маршрутов с указанием модели самолета, выполняющего рейсы по этим маршрутам. Воспользуемся таблицами «Маршруты» (routes) и «Самолеты» (aircrafts).

```
EXPLAIN
  SELECT r.flight_no, r.departure_airport_name,
         r.arrival_airport_name, a.model
    FROM routes r
    JOIN aircrafts a ON r.aircraft_code = a.aircraft_code
    ORDER BY flight_no;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Sort (cost=24.25..24.31 rows=21 width=124)
   Sort Key: r.flight_no
   -> Hash Join (cost=1.20..23.79 rows=21 width=124)
        Hash Cond: (r.aircraft_code = a.aircraft_code)
        -> Seq Scan on routes r (cost=0.00..20.64 rows=464 width=108)
        -> Hash (cost=1.09..1.09 rows=9 width=48)
             -> Seq Scan on aircrafts a (cost=0.00..1.09 rows=9 width=48)
(7 строк)
```

На самом внутреннем уровне плана последовательно сканируется (Seq Scan) таблица aircrafts, и формируется хеш-таблица, ключами которой являются значения атрибута aircraft_code, т. к. именно по нему выполняется соединение таблиц. Затем последовательно сканируется (Seq Scan) таблица routes, и для каждой ее строки выполняется поиск значения атрибута aircraft_code среди ключей хеш-таблицы: Hash Cond: (r.aircraft_code = a.aircraft_code). Если такой поиск успешен, значит, формируется комбинированная результирующая строка выборки. На верхнем уровне плана сформированные строки сортируются. Обратите внимание, что хеш-таблица создана на основе той таблицы, число строк в которой меньше, т. е. aircrafts. Таким образом, поиск в ней будет выполняться быстрее, чем если бы хеш-таблица была создана на основе таблицы routes.

Последний из методов соединения наборов строк — **соединение слиянием** (merge join). Для иллюстрации воспользуемся простым запросом, построенным на основе таблиц «Билеты» (tickets) и «Перелеты» (ticket_flights). Он выбирает для каждого билета все перелеты, включенные в него. Конечно, это очень упрощенный запрос, в реальной ситуации он не представлял бы большой практической пользы, но в целях упрощения плана и повышения наглядности, воспользуемся им.

```
EXPLAIN
  SELECT t.ticket_no, t.passenger_name,
         tf.flight_id, tf.amount
    FROM tickets t
    JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
    ORDER BY t.ticket_no;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Merge Join (cost=1.51..98276.90 rows=1045726 width=40)
    Merge Cond: (t.ticket_no = tf.ticket_no)
    -> Index Scan using tickets_pkey on tickets t (cost=0.42..17230.42
 ↪  rows=366733 width=30)
    -> Index Scan using ticket_flights_pkey on ticket_flights tf
 ↪  (cost=0.42..67058.74 rows=1045726 width=24)
(4 строки)
```

Два внутренних узла дерева плана отвечают за сканирование таблиц tickets и ticket_flights по индексам (Index Scan). Таким образом, верхний узел (Merge Join) получает наборы строк этих таблиц уже в отсортированном виде, поэтому не требуется отдельного узла для сортировки результирующих строк. Обратите внимание на оценки стоимости выполнения всех трех операций: двух сканирований таблиц и результирующего соединения слиянием. Мы видим, что первая оценка в узле Merge Join

равна $1,51$, что значительно меньше вторых оценок, вычисленных планировщиком для двух нижних узлов, а именно: $17230,42$ и $67058,74$. Напомним, что первая оценка говорит, сколько ресурсов будет затрачено (сколько времени, в условных единицах, пройдет) до начала вывода первых результатов выполнения операции на данном уровне дерева плана. Вторая оценка показывает общее количество ресурсов, требующихся для полного завершения операции на данном уровне дерева плана. Таким образом, можно заключить, что вывод результирующих строк начнется еще задолго до завершения сканирования исходных таблиц.

## 10.4 Управление планировщиком

Для управления планировщиком предусмотрен целый ряд параметров. Их можно изменить на время текущего сеанса работы с помощью команды SET. Конечно, изменять параметры в производственной базе данных следует только в том случае, когда вы *обоснованно* считаете, что планировщик ошибается. Однако для того чтобы научиться видеть ошибки планировщика, нужен большой опыт. Поэтому следует рассматривать приведенные далее команды управления планировщиком лишь с позиции изучения потенциальных возможностей управления им, а не как рекомендацию к бездумному изменению этих параметров в реальной работе.

Например, чтобы запретить планировщику использовать метод соединения на основе хеширования, нужно сделать так:

```
SET enable_hashjoin = off;
```

Чтобы запретить планировщику использовать метод соединения слиянием, нужно сделать так:

```
SET enable_mergejoin = off;
```

А для того чтобы запретить планировщику использовать соединение методом вложенного цикла, нужно сделать так:

```
SET enable_nestloop = off;
```

По умолчанию все эти параметры имеют значение «on» (включено).

Необходимо уточнить, что в результате выполнения вышеприведенных команд не накладывается полного запрета на использование конкретного метода соединения наборов строк. Методу просто назначается очень высокая стоимость, но планировщик все равно сохраняет возможность маневра, и даже такой «запрещенный» метод может быть использован. Более подробно этот вопрос рассматривается в одном из примеров в разделе «Контрольные вопросы и задания».

Давайте запретим планировщику использовать метод соединения слиянием:

```
SET enable_mergejoin = off;
```

```
SET
```

Теперь повторим предыдущий запрос:

```
EXPLAIN
  SELECT t.ticket_no, t.passenger_name,
         tf.flight_id, tf.amount
    FROM tickets t
    JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
    ORDER BY t.ticket_no;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Sort (cost=226400.55..229014.87 rows=1045726 width=40)
   Sort Key: t.ticket_no
   -> Hash Join (cost=16824.49..64658.49 rows=1045726 width=40)
        Hash Cond: (tf.ticket_no = t.ticket_no)
        -> Seq Scan on ticket_flights tf (cost=0.00..18692.26
 ↪  rows=1045726 width=24)
        -> Hash (cost=9733.33..9733.33 rows=366733 width=30)
             -> Seq Scan on tickets t (cost=0.00..9733.33 rows=366733
 ↪  width=30)
(7 строк)
```

Теперь планировщик выбирает слияние хешированием. Полученные оценки стоимости выполнения запроса будут значительно выше. При этом вывод результирующих строк начнется значительно позднее, чем при использовании метода соединения слиянием. На это указывает значение параметра cost для верхнего узла дерева плана — cost=226400.55..229014.87.

В команде EXPLAIN можно указать опцию ANALYZE, что позволит выполнить запрос и вывести на экран фактические затраты времени на выполнение запроса и число фактически выбранных строк. При этом, хотя запрос и выполняется, сами результирующие строки не выводятся.

Сначала разрешим планировщику использовать метод соединения слиянием:

```
SET enable_mergejoin = on;
```

```
SET
```

Повторим предыдущий запрос с опцией ANALYZE.

```
EXPLAIN ANALYZE
  SELECT t.ticket_no, t.passenger_name,
         tf.flight_id, tf.amount
    FROM tickets t
    JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
    ORDER BY t.ticket_no;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Merge Join (cost=1.51..98276.90 rows=1045726 width=40)
            (actual time=0.087..10642.643 rows=1045726 loops=1)
   Merge Cond: (t.ticket_no = tf.ticket_no)
   -> Index Scan using tickets_pkey on tickets t
            (cost=0.42..17230.42 rows=366733 width=30)
            (actual time=0.031..762.460 rows=366733 loops=1)
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
            (cost=0.42..67058.74 rows=1045726 width=24)
```

```
           (actual time=0.006..7743.705 rows=1045726 loops=1)
 Planning time: 122.347 ms
 Execution time: 10948.791 ms
(6 строк)
```

Фактические затраты времени измеряются в миллисекундах, а оценки стоимости — в условных единицах, поэтому плановые оценки и фактические затраты совпасть не могут. Важнее обратить внимание на то, насколько точно планировщик оценил число обрабатываемых строк, а также на фактическое число повторений того или иного узла дерева плана — это параметр loops. В данном запросе каждый узел плана был выполнен ровно один раз, поскольку выбор строк из обоих соединяемых наборов производился по индексу, поэтому достаточно одного прохода по каждому набору. Число выбираемых строк было оценено точно, поскольку таблицы связаны по внешнему ключу, и в выборку включаются все их строки (нет предложения WHERE).

Кроме времени выполнения запроса выводится также и время формирования плана.

Необходимо учитывать, что фактические затраты времени на разных компьютерах будут различаться. Будет другим и фактическое время при повторном выполнении запроса на одном и том же компьютере, поскольку и в СУБД, и в операционной системе используются буферизация и кэширование, а также с течением времени может изменяться фактическая нагрузка на сервер. Поэтому время выполнения повторного запроса может оказаться значительно меньше, чем время выполнения первого, а время выполнения запроса в третий раз — немного больше, чем во второй.

Если модифицировать запрос, добавив предложение WHERE, то точного совпадения оценки числа выбираемых строк и фактического их числа уже не будет.

```
EXPLAIN ANALYZE
  SELECT t.ticket_no, t.passenger_name,
         tf.flight_id, tf.amount
    FROM tickets t
    JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
    WHERE amount > 50000
    ORDER BY t.ticket_no;
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Merge Join (cost=27391.09..46664.80 rows=75126 width=40)
            (actual time=2133.715..3117.200 rows=72647 loops=1)
    Merge Cond: (t.ticket_no = tf.ticket_no)
    -> Index Scan using tickets_pkey on tickets t
            (cost=0.42..17230.42 rows=366733 width=30)
            (actual time=0.009..318.517 rows=366733 loops=1)
    -> Sort (cost=27390.66..27578.48 rows=75126 width=24)
            (actual time=2132.781..2173.526 rows=72647 loops=1)
         Sort Key: tf.ticket_no
         Sort Method: external sort Disk: 2768kB
         -> Seq Scan on ticket_flights tf
                (cost=0.00..21306.58 rows=75126 width=24)
                (actual time=0.351..332.313 rows=72647 loops=1)
            Filter: (amount > '50000'::numeric)
            Rows Removed by Filter: 973079
```

```
 Planning time: 1.415 ms
 Execution time: 3135.869 ms
(11 строк)
```

План выполнения запроса изменился. Метод соединения наборов строк остался прежним — слияние. Но выборка строк в нижнем узле дерева плана теперь выполняется с помощью последовательного сканирования и сортировки. Обратите внимание, что при включении опции ANALYZE может выводиться дополнительная информация о фактически использовавшихся методах, о затратах памяти и др. В частности, сказано, что была использована внешняя сортировка на диске, приведены затраты памяти на ее выполнение, приведено число строк, удаленных при проверке условия их отбора:

```
 Sort Method: external sort Disk: 2768kB
 Rows Removed by Filter: 973079
```

Фактическое число строк, выбранных из таблицы ticket_flights, и фактическое число результирующих строк запроса хотя и не совпали с оценками этих чисел, но оказались весьма близкими к ним. Фактические значения равны 72647, а оценки — 75126. Это хороший результат работы планировщика.

Обратимся еще раз к запросу, который мы уже рассматривали выше, и выполним его с опцией ANALYZE. В плане этого запроса нас будет интересовать фактический параметр loops.

```
EXPLAIN ANALYZE
  SELECT a.aircraft_code, a.model,
         s.seat_no, s.fare_conditions
    FROM seats s
    JOIN aircrafts a ON s.aircraft_code = a.aircraft_code
    WHERE a.model ~ '^Air'
    ORDER BY s.seat_no;
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Sort (cost=23.28..23.65 rows=149 width=59)
      (actual time=3.423..3.666 rows=426 loops=1)
    Sort Key: s.seat_no
    Sort Method: quicksort Memory: 46kB
    -> Nested Loop (cost=5.43..17.90 rows=149 width=59)
                   (actual time=0.236..0.993 rows=426 loops=1)
         -> Seq Scan on aircrafts a (cost=0.00..1.11 rows=1 width=48)
                   (actual time=0.100..0.112 rows=3 loops=1)
              Filter: (model ~ '^Air'::text)
              Rows Removed by Filter: 6
         -> Bitmap Heap Scan on seats s (cost=5.43..15.29 rows=149
 ↪  width=15)
                   (actual time=0.080..0.154 rows=142 loops=3)
              Recheck Cond: (aircraft_code = a.aircraft_code)
              Heap Blocks: exact=6
              -> Bitmap Index Scan on seats_pkey (cost=0.00..5.39
 ↪  rows=149 width=0)
                   (actual time=0.064..0.064 rows=142 loops=3)
                   Index Cond: (aircraft_code = a.aircraft_code)
 Planning time: 0.554 ms
```

```
 Execution time: 3.840 ms
(14 строк)
```

Как видно из плана, значение параметра loops для узла, выполняющего сканирование таблицы seats по индексу с построением битовой карты, равно трем. Это объясняется тем, что из таблицы aircrafts были фактически выбраны три строки, и *для каждой* из них выполняется поиск в таблице seats. Для подсчета общих затрат времени на выполнение операций сканирования по индексу за три цикла нужно значение параметра actual time умножить на значение параметра loops. Таким образом, для узла дерева плана Bitmap Index Scan получим:

$$0,064 \times 3 = 0,192.$$

Подобные вычисления общих затрат времени на промежуточных уровнях дерева плана могут помочь выявить наиболее ресурсоемкие операции.

Попутно заметим, что, согласно этому плану, сортировка на верхнем уровне плана выполнялась в памяти с использованием метода quicksort:

```
Sort Method: quicksort Memory: 46kB
```

До сих пор мы рассматривали только выборки, т. е. такие запросы, которые не изменяют хранимых данных. Однако, кроме выборок, есть такие операции, как вставка, обновление и удаление строк. Нужно помнить, что хотя результаты выборки и не выводятся, тем не менее, она фактически все равно выполняется. Поэтому если требуется исследовать план выполнения запроса, модифицирующего данные, то для того, чтобы изменения на самом деле произведены не были, нужно воспользоваться транзакцией с откатом изменений.

```
BEGIN;
```

```
BEGIN
```

```
EXPLAIN ANALYZE
  UPDATE aircrafts
    SET range = range + 100
    WHERE model ~ '^Air';
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Update on aircrafts (cost=0.00..1.11 rows=1 width=58)
                     (actual time=0.299..0.299 rows=0 loops=1)
    -> Seq Scan on aircrafts (cost=0.00..1.11 rows=1 width=58)
                     (actual time=0.111..0.121 rows=3 loops=1)
         Filter: (model ~ '^Air'::text)
         Rows Removed by Filter: 6
 Planning time: 0.235 ms
 Execution time: 0.414 ms
(6 строк)
```

```
ROLLBACK;
```

```
ROLLBACK
```

В документации приводится важное предостережение о том, что нельзя экстраполировать, т. е. распространять, пусть даже и с некоторыми поправками, оценки, полученные для таблиц небольшого размера, на таблицы большого размера. Это объясняется тем, что оценки, вычисляемые планировщиком, не являются линейными. Одна из причин заключается в том, что для таблиц разных размеров могут быть выбраны разные планы. Например, для маленькой таблицы может быть выбрано последовательное сканирование, а для большой — сканирование по индексу.

## 10.5 Оптимизация запросов

Мы рассмотрели базовые способы получения плана выполнения запроса и познакомились с типичными компонентами плана. Эти знания призваны помочь в тех ситуациях, когда необходимо ускорить выполнение запроса. При принятии решения о том, что выполнение какого-либо запроса нужно оптимизировать, следует учитывать не только абсолютное время его выполнения, но и частоту его использования. Запрос может выполняться, например, за несколько миллисекунд, но таких запросов могут быть сотни или тысячи.

В результате ресурсы сервера будут расходоваться очень интенсивно. Возможно, что в такой ситуации придется заняться ускорением выполнения этого запроса. А если запрос выполняется один раз в месяц, скажем, для получения итоговой картины по продажам авиабилетов за этот период, то в этом случае бороться за ускорение на несколько миллисекунд, видимо, не имеет смысла.

Повлиять на скорость выполнения запроса можно различными способами, мы рассмотрим некоторые из них:

- обновление статистики, на основе которой планировщик строит планы;
- изменение исходного кода запроса;
- изменение схемы данных, связанное с денормализацией: создание материализованных представлений и временных таблиц, создание индексов, использование вычисляемых столбцов таблиц;
- изменение параметров планировщика, управляющих выбором порядка соединения наборов строк: использование общих табличных выражений (запросы с предложением WITH), использование фиксированного порядка соединения (параметр join_collapse_limit = 1), запрет раскрытия подзапросов и преобразования их в соединения таблиц (параметр from_collapse_limit = 1);
- изменение параметров планировщика, управляющих выбором метода доступа (enable_seqscan, enable_indexscan, enable_indexonlyscan, enable_bitmapscan) и способа соединения наборов строк (enable_nestloop, enable_hashjoin, enable_mergejoin);
- изменение параметров планировщика, управляющих использованием ряда операций: агрегирование на основе хеширования (enable_hashagg), материализация временных наборов строк (enable_material), выполнение явной сортировки при наличии других возможностей (enable_sort).

Необходимым условием для того, чтобы планировщик выбрал правильный план, является наличие актуальной статистики. Если вы предполагаете, что планировщик опирается на неактуальную статистику, можно ее принудительно обновить с помощью команды ANALYZE. Например, обновить статистику для таблицы aircrafts можно, выполнив команду

```
ANALYZE aircrafts;
```

```
ANALYZE
```

В качестве примера ситуации, в которой оптимизация запроса представляется обоснованной, рассмотрим следующую задачу. Предположим, что необходимо определить степень загруженности кассиров нашей авиакомпании в сентябре 2016 г. Для этого, в частности, требуется выявить распределение числа операций бронирования по числу билетов, оформленных в рамках этих операций. Другими словами, это означает, что нужно подсчитать число операций бронирования, в которых был оформлен только один билет, число операций, в которых было оформлено два билета и т. д.

Эту задачу можно переформулировать так: для каждой строки, отобранной из таблицы «Бронирования» (bookings), нужно подсчитать соответствующие строки в таблице «Билеты» (tickets). Речь идет о строках, в которых значение поля book_ref такое же, что и в текущей строке таблицы bookings. Буквальное следование такой формулировке задачи приводит к получению запроса с коррелированным подзапросом в предложении SELECT. Но это еще не окончательное решение. Теперь нужно сгруппировать полученный набор строк по значениям числа оформленных билетов.

Получаем такой запрос:

```
EXPLAIN
  SELECT num_tickets, count( * ) AS num_bookings
    FROM (
      SELECT b.book_ref,
        ( SELECT count( * ) FROM tickets t
            WHERE t.book_ref = b.book_ref
        )
        FROM bookings b
        WHERE date_trunc( 'mon', book_date ) = '2016-09-01'
    ) AS count_tickets( book_ref, num_tickets )
    GROUP by num_tickets
    ORDER BY num_tickets DESC;
```

```
                          QUERY PLAN
---------------------------------------------------------------------
 GroupAggregate (cost=14000017.12..27994373.35 rows=1314 width=16)
    Group Key: ((SubPlan 1))
    -> Sort (cost=14000017.12..14000020.40 rows=1314 width=8)
         Sort Key: ((SubPlan 1)) DESC
         -> Seq Scan on bookings b
            (cost=0.00..13999949.05 rows=1314 width=8)
             Filter: (date_trunc('mon'::text, book_date) =
                      '2016-09-01 00:00:00+08'::timestamp with time zone)
             SubPlan 1
              -> Aggregate (cost=10650.17..10650.18 rows=1 width=8)
                   -> Seq Scan on tickets t
                      (cost=0.00..10650.16 rows=2 width=0)
```

```
                        Filter: (book_ref = b.book_ref)
(10 строк)
```

В этом плане получены очень большие оценки общей стоимости выполнения запроса: cost=14000017.12..27994373.35. Универсальной зависимости между оценкой стоимости и реальным временем выполнения запроса не существует. Не всегда можно даже приблизительно предположить, в какие затраты времени выльется та или иная оценка стоимости. Но, тем не менее, в тексте главы при рассмотрении других запросов оценок такого порядка еще не встречалось. Планировщик предполагает, что из таблицы tickets в подзапросе будет извлекаться всего по две строки, и эту операцию нужно будет проделать 1314 раз: столько строк предположительно будет выбрано из таблицы bookings. Как видно из плана, для просмотра строк в таблице tickets используется ее последовательное сканирование. В результате оценка стоимости этого узла плана получается высокой — cost=0.00..10650.16.

Если у вас не очень мощный компьютер, то время получения результата может выйти за разумные пределы, и вам придется прервать процесс с помощью клавишей Ctrl-C.

Что можно сделать для ускорения выполнения запроса? Давайте создадим индекс для таблицы tickets по столбцу book_ref, по которому происходит поиск в ней.

```
CREATE INDEX tickets_book_ref_key ON tickets ( book_ref );
```

```
CREATE INDEX
```

Повторим запрос, добавив параметр ANALYZE в команду EXPLAIN. Новый план, в котором отражены и фактические результаты, будет таким:

```
                              QUERY PLAN
-----------------------------------------------------------------------
 GroupAggregate (cost=22072.70..38484.52 rows=1314 width=16)
                (actual time=3656.554..3787.562 rows=5 loops=1)
    Group Key: ((SubPlan 1))
    -> Sort (cost=22072.70..22075.99 rows=1314 width=8)
           (actual time=3656.533..3726.969 rows=165534 loops=1)
        Sort Key: ((SubPlan 1)) DESC
        Sort Method: external merge Disk: 2912kB
        -> Seq Scan on bookings b
           (cost=0.00..22004.64 rows=1314 width=8)
           (actual time=0.219..3332.162 rows=165534 loops=1)
           Filter: (date_trunc('mon'::text, book_date) =
                    '2016-09-01 00:00:00+08'::timestamp with time zone)
           Rows Removed by Filter: 97254
           SubPlan 1
            -> Aggregate (cost=12.46..12.47 rows=1 width=8)
               (actual time=0.016..0.016 rows=1 loops=165534)
               -> Index Only Scan using tickets_book_ref_key on tickets t
                  (cost=0.42..12.46 rows=2 width=0)
                  (actual time=0.013..0.014 rows=1 loops=165534)
                    Index Cond: (book_ref = b.book_ref)
                    Heap Fetches: 230699
 Planning time: 0.290 ms
 Execution time: 3788.690 ms
(15 строк)
```

Теперь планировщик использует индекс для поиска в таблице tickets. Причем, это поиск исключительно по индексу (Index Only Scan), поскольку нас интересует только число строк — count( * ), а не их содержание. Обратите внимание на различие предполагаемого и фактического числа извлекаемых строк. Тем не менее, запрос стал выполняться значительно — на порядок — быстрее.

Результат имеет такой вид:

```
 num_tickets | num_bookings
-------------+--------------
           5 |           13
           4 |          536
           3 |         7966
           2 |        47573
           1 |       109446
(5 строк)
```

Кроме создания индекса есть и другой способ: замена коррелированного подзапроса соединением таблиц.

```
EXPLAIN ANALYZE
  SELECT num_tickets, count( * ) AS num_bookings
     FROM (
       SELECT b.book_ref, count( * )
         FROM bookings b, tickets t
         WHERE date_trunc( 'mon', b.book_date ) = '2016-09-01'
           AND t.book_ref = b.book_ref
         GROUP BY b.book_ref
     ) AS count_tickets( book_ref, num_tickets )
     GROUP by num_tickets
     ORDER BY num_tickets DESC;
```

```
                            QUERY PLAN
------------------------------------------------------------------------
GroupAggregate (cost=16966.67..16978.53 rows=200 width=16)
               (actual time=4092.258..4219.477 rows=5 loops=1)
  Group Key: count_tickets.num_tickets
  -> Sort (cost=16966.67..16969.96 rows=1314 width=8)
          (actual time=4092.236..4161.294 rows=165534 loops=1)
     Sort Key: count_tickets.num_tickets DESC
     Sort Method: external merge Disk: 2912kB
     -> Subquery Scan on count_tickets
        (cost=16858.57..16898.61 rows=1314 width=8)
        (actual time=3176.113..3862.133 rows=165534 loops=1)
        -> GroupAggregate
           (cost=16858.57..16885.47 rows=1314 width=15)
           (actual time=3176.111..3765.157 rows=165534 loops=1)
             Group Key: b.book_ref
           -> Sort (cost=16858.57..16863.16 rows=1834 width=7)
              (actual time=3176.098..3552.334 rows=230699 loops=1)
              Sort Key: b.book_ref
              Sort Method: external merge Disk: 3824kB
              -> Hash Join (cost=5632.24..16759.16 rows=1834 width=7)
                 (actual time=498.701..1091.509 rows=230699 loops=1)
                  Hash Cond: (t.book_ref = b.book_ref)
```

```
                  -> Seq Scan on tickets t
                     (cost=0.00..9733.33 rows=366733 width=7)
                     (actual time=0.047..170.792 rows=366733 loops=1)
                      -> Hash (cost=5615.82..5615.82 rows=1314 width=7)
                         (actual time=498.624..498.624 rows=165534
↪  loops=1)
                         Buckets: 262144 (originally 2048)
                         Batches: 2 (originally 1)
                         Memory Usage: 3457kB
                         -> Seq Scan on bookings b
                            (cost=0.00..5615.82 rows=1314 width=7)
                            (actual time=0.019..267.728 rows=165534
↪  loops=1)
                            Filter: (date_trunc('mon'::text, book_date) =
                                     '2016-09-01 00:00:00+08'::timestamp
↪  with time zone)
                            Rows Removed by Filter: 97254
 Planning time: 2.183 ms
 Execution time: 4221.133 ms
(21 строка)
```

В данном плане используется соединение хешированием (Hash Join). При этом индекс по таблице tickets игнорируется: она просматривается последовательно (Seq Scan on tickets t). Время выполнения модифицированного запроса оказывается несколько большим, чем в предыдущем случае, когда в запросе присутствовал коррелированный подзапрос. Таким образом, можно заключить, что для ускорения работы оригинального запроса можно было либо создать индекс, либо модифицировать сам запрос, даже не создавая индекса.

Другие методы оптимизации выполнения запросов представлены в разделе «Контрольные вопросы и задания». Рекомендуем вам самостоятельно с ними ознакомиться и поэкспериментировать.

Перед выполнением упражнений нужно восстановить измененные значения параметров:

```
SET enable_hashjoin = on;
```

```
SET
```

```
SET enable_nestloop = on;
```

```
SET
```

## Контрольные вопросы и задания

1. Как вы думаете, почему при сканировании по индексу оценка стоимости ресурсов, требующихся для выдачи первых результатов, не равна нулю, хотя используется индекс, совпадающий с порядком сортировки?

   ```
   EXPLAIN SELECT * FROM bookings ORDER BY book_ref;
   ```

```
                              QUERY PLAN
-------------------------------------------------------------------
 Index Scan using bookings_pkey on bookings (cost=0.42..8511.24
 ↪  rows=262788 width=21)
(1 строка)
```

2. Как вы думаете, если в запросе присутствует предложение ORDER BY, и создан индекс по тем столбцам, которые фигурируют в предложении ORDER BY, то всегда ли будет использоваться этот индекс или нет? Почему? Проверьте ваши предположения с помощью команды EXPLAIN.

3. Самостоятельно выполните команду EXPLAIN для запроса, содержащего общее табличное выражение (CTE). Посмотрите, на каком уровне находится узел плана, отвечающий за это выражение, как он оформляется. Учтите, что общие табличные выражения всегда материализуются, т. е. вычисляются однократно и результат их вычисления сохраняется в памяти, а затем все последующие обращения в рамках запроса направляются уже к этому материализованному результату.

4. Прокомментируйте следующий план, попробуйте объяснить значения всех его узлов и параметров.

```
EXPLAIN
  SELECT total_amount FROM bookings
  ORDER BY total_amount DESC LIMIT 5;
```

```
                              QUERY PLAN
-------------------------------------------------------------------
 Limit (cost=8666.69..8666.71 rows=5 width=6)
   -> Sort (cost=8666.69..9323.66 rows=262788 width=6)
        Sort Key: total_amount DESC
        -> Seq Scan on bookings (cost=0.00..4301.88 rows=262788
 ↪  width=6)
(4 строки)
```

5. В подавляющем большинстве городов только один аэропорт, но есть и такие города, в которых более одного аэропорта. Давайте их выявим.

```
EXPLAIN
  SELECT city, count( * ) FROM airports
  GROUP BY city HAVING count( * ) > 1;
```

```
                              QUERY PLAN
-------------------------------------------------------------------
 HashAggregate (cost=3.82..4.83 rows=101 width=25)
   Group Key: city
   Filter: (count(*) > 1)
     -> Seq Scan on airports (cost=0.00..3.04 rows=104 width=17)
(4 строки)
```

Для подсчета числа аэропортов в городах используется последовательное сканирование и формирование хеш-таблицы (HashAggregate), ключом которой является столбец city. Затем из нее отбираются те записи, значения которых соответствуют условию

```
Filter: (count(*) > 1)
```

Как вы думаете, чем можно объяснить, что вторая оценка стоимости в параметре cost для узла Seq Scan, равная 3.04, не совпадает с первой оценкой стоимости в параметре cost для узла HashAggregate?

6. Выполните команду EXPLAIN для запроса, в котором использована одна из оконных функций. Найдите в плане выполнения запроса узел с именем WindowAgg. Попробуйте объяснить, почему он занимает именно этот уровень в плане.

7. Выполните анализ плана выполнения операций вставки и удаления строк. Причем, сделайте это таким образом, чтобы данные в таблицах фактически изменены не были.

8.* Замена коррелированного подзапроса соединением таблиц является одним из способов повышения производительности.

   Предположим, что мы задались вопросом: сколько маршрутов обслуживают самолеты каждого типа? При этом нужно учитывать, что может иметь место такая ситуация, когда самолеты какого-либо типа не обслуживают ни одного маршрута. Поэтому необходимо использовать не только представление «Маршруты» (routes), но и таблицу «Самолеты» (aircrafts).

   Это первый вариант запроса, в нем используется коррелированный подзапрос.

```
EXPLAIN ANALYZE
  SELECT a.aircraft_code AS a_code,
         a.model,
         ( SELECT count( r.aircraft_code ) FROM routes r
             WHERE r.aircraft_code = a.aircraft_code
         ) AS num_routes
    FROM aircrafts a
    GROUP BY 1, 2
    ORDER BY 3 DESC;
```

   А в этом варианте коррелированный подзапрос раскрыт и заменен внешним соединением. Причина использования внешнего соединения в том, что может найтись модель самолета, не обслуживающая ни одного маршрута, и если не использовать внешнее соединение, она вообще не попадет в результирующую выборку.

```
EXPLAIN ANALYZE
  SELECT a.aircraft_code AS a_code,
         a.model,
         count( r.aircraft_code ) AS num_routes
    FROM aircrafts a
    LEFT OUTER JOIN routes r
      ON r.aircraft_code = a.aircraft_code
    GROUP BY 1, 2
    ORDER BY 3 DESC;
```

   Исследуйте планы выполнения обоих запросов. Попытайтесь найти объяснение различиям в эффективности их выполнения. Чтобы получить усредненную картину, выполните каждый запрос несколько раз. Поскольку таблицы, участвующие в запросах, небольшие, то различие по абсолютным затратам времени вы-

полнения будет незначительным. Но если бы число строк в таблицах было большим, то экономия ресурсов сервера могла оказаться заметной.

Предложите аналогичную пару запросов к базе данных «Авиаперевозки». Проведите необходимые эксперименты с вашими запросами.

9. Одним из способов повышения производительности является изменение схемы данных, связанное с денормализацией, а именно: создание материализованных представлений. В главе 5 было описано такое материализованное представление — «Маршруты» (routes). Команда для его создания была приведена в главе 6.

   Проведите эксперимент: сначала выполните выборку из готового представления, а затем ту выборку, которая это представление формирует.

   ```
   EXPLAIN ANALYZE SELECT * FROM routes;

   EXPLAIN ANALYZE WITH f3 AS ( SELECT f2.flight_no, ...
   ```

   Поскольку второй запрос очень громоздкий, то можно поступить таким образом: сначала сохраните его в текстовом файле, а затем выполните с помощью команды \i утилиты psql.

   Вы увидите, что затраты времени отличаются практически на два порядка. Конечно, нужно помнить, что материализованные представления необходимо периодически обновлять, чтобы их содержимое было актуальным.

10.* Одним из способов повышения производительности является изменение схемы данных, связанное с денормализацией, а именно: использование вычисляемых столбцов таблиц. В качестве примера рассмотрим таблицу «Бронирования» (bookings). В ней столбец «Полная сумма бронирования» (total_amount) является вычисляемым. Мы не будем сейчас говорить о том, каким образом его значения синхронизируются с данными в таблице «Перелеты» (ticket_flifgts), а лишь рассмотрим два запроса, возвращающие полные суммы бронирований. Предположим, что указанного столбца в таблице bookings не было бы. Тогда запрос, возвращающий полные суммы бронирований, выглядел бы так:

   ```
   EXPLAIN ANALYZE
     SELECT b.book_ref, sum( tf.amount )
     FROM bookings b, tickets t, ticket_flights tf
     WHERE b.book_ref = t.book_ref
       AND t.ticket_no = tf.ticket_no
     GROUP BY 1 ORDER BY 1;
   ```

   Но благодаря наличию вычисляемого столбца total_amount те же сведения можно получить с гораздо меньшими затратами ресурсов:

   ```
   EXPLAIN ANALYZE
     SELECT book_ref, total_amount FROM bookings ORDER BY 1;
   ```

   Попробуйте предложить еще какой-нибудь вычисляемый столбец для одной из таблиц базы данных «Авиаперевозки». Проведите эксперименты, подтверждающие эффективность вашего решения.

11.* Одним из способов повышения производительности является изменение схемы данных, а именно: использование временных таблиц. Предположим, что нам предстоит сделать много выборок из представления «Рейсы» (flights_v), в таком случае имеет смысл подумать о создании временной таблицы:

```
CREATE TEMP TABLE flights_tt AS SELECT * FROM flights_v;
```

Сформируйте планы для получения простой выборки из представления и из временной таблицы и сравните полученные результаты. Как вы думаете, почему план, сформированный для получения даже простой выборки из представления, многоуровневый?

```
EXPLAIN ANALYZE SELECT * FROM flights_v;
```

```
EXPLAIN ANALYZE SELECT * FROM flights_tt;
```

Выполните более сложные запросы к представлению и временной таблице и сравните полученные результаты. Включайте в команду EXPLAIN опцию ANALYZE, чтобы увидеть фактические затраты времени.

Подумайте, при выполнении каких запросов к базе данных «Авиаперевозки» было бы целесообразно создать временную таблицу. Создайте ее и проведите эксперименты, подтверждающие эффективность ее использования.

12. Одним из способов повышения производительности является изменение схемы данных, связанное с денормализацией, а именно: создание индексов.

Выполните следующий простой запрос к таблице «Билеты» (tickets):

```
EXPLAIN ANALYZE
  SELECT count( * ) FROM tickets
  WHERE passenger_name = 'IVAN IVANOV';
```

Создайте индекс по столбцу passenger_name:

```
CREATE INDEX passenger_name_key ON tickets ( passenger_name );
```

Теперь повторите запрос и сравните полученные планы и фактические результаты.

Предложите какой-нибудь запрос к базе данных «Авиаперевозки», для выполнения которого было бы целесообразно создать индекс. Создайте индекс и повторите запрос. Изучите полученный план, посмотрите, используется ли индекс планировщиком.

13.* В самом конце главы мы выполняли оптимизацию запроса путем создания индекса и модификации текста запроса. Был сформирован такой запрос:

```
EXPLAIN ANALYZE
  SELECT num_tickets, count( * ) AS num_bookings
    FROM (
      SELECT b.book_ref, count( * )
        FROM bookings b, tickets t
        WHERE date_trunc( 'mon', b.book_date ) = '2016-09-01'
          AND t.book_ref = b.book_ref
        GROUP BY b.book_ref
    ) AS count_tickets( book_ref, num_tickets )
```

```
  GROUP by num_tickets
  ORDER BY num_tickets DESC;
```

Были проведены два эксперимента с параметрами планировщика enable_hashjoin и enable_nestloop при наличии индекса по таблице tickets.

```
SET enable_hashjoin = off;
```

```
SET
```

```
SET enable_nestloop = off;
```

```
SET
```

Однако полученные планы детально рассмотрены не были.

**Задание.** Проанализируйте эти планы. Посмотрите, в каких случаях используются и в каких не используются индексы по таблицам bookings и tickets. Вспомните о таком понятии, как *селективность*, т. е. доля строк, выбираемых из таблицы.

14. В столбцах таблиц могут содержаться значения NULL. При сортировке строк по значениям таких столбцов СУБД по умолчанию ведет себя так, как будто значение NULL превосходит по величине любые другие значения. В результате получается, что если задан возрастающий порядок сортировки, то значения NULL будут идти последними, если же порядок сортировки убывающий, тогда они будут первыми. Принимая решение о создании индексов, нужно учитывать требуемый порядок сортировки и желаемое расположение строк со значениями NULL в выборке (в ее начале или в конце).

    Давайте создадим таблицу, содержащую такое число строк, что использование индекса планировщиком становится очень вероятным, и выполним с этой таблицей ряд экспериментов.

```
CREATE TABLE nulls AS
  SELECT num::integer, 'TEXT' || num::text AS txt
    FROM generate_series( 1, 200000 ) AS gen_ser( num );
```

```
SELECT 200000
```

Проиндексируем таблицу по числовому столбцу. Поскольку мы не указываем порядок сортировки явным образом, то по умолчанию он будет возрастающим, как если бы мы использовали ключевое слово ASC.

```
CREATE INDEX nulls_ind ON nulls ( num );
```

```
CREATE INDEX
```

Добавим в таблицу одну строку, содержащую значение NULL в индексируемом столбце:

```
INSERT INTO nulls VALUES ( null, 'TEXT' );
```

```
INSERT 0 1
```

Теперь посмотрим, будет ли использоваться индекс в следующем запросе:

```
EXPLAIN
  SELECT * FROM nulls ORDER BY num;
```

Да, индекс используется.

```
                              QUERY PLAN
-----------------------------------------------------------------
 Index Scan using nulls_ind on nulls  (cost=0.42..5852.42 rows=200000
 ↪  width=14)
(1 строка)
```

Вы можете убедиться, что строка со значением NULL окажется в выводе самой последней. Поскольку в нашей таблице очень много строк, воспользуемся предложением OFFSET:

```
SELECT * FROM nulls ORDER BY num OFFSET 199995;
```

```
  num   |    txt
--------+------------
 199996 | TEXT199996
 199997 | TEXT199997
 199998 | TEXT199998
 199999 | TEXT199999
 200000 | TEXT200000
        | TEXT
(6 строк)
```

Модифицируем запрос, явно указав, что значения NULL должны располагаться в самом начале выборки, и посмотрим, будет ли использоваться индекс теперь.

```
EXPLAIN
  SELECT * FROM nulls ORDER BY num NULLS FIRST;
```

Теперь индекс не используется. Вместо этого выполняется последовательное сканирование таблицы и сортировка выбранных строк.

```
                                QUERY PLAN
-------------------------------------------------------------------
 Sort  (cost=24110.14..24610.14 rows=200000 width=14)
   Sort Key: num NULLS FIRST
   ->  Seq Scan on nulls  (cost=0.00..3081.00 rows=200000 width=14)
(3 строки)
```

**Задание 1.** Ниже приведена модифицированная команды выборки из таблицы nulls. Не выполняя эту команду, попытайтесь ответить, будет ли использоваться созданный нами индекс при выполнении такой выборки, а затем проверьте вашу гипотезу на практике.

```
EXPLAIN
 SELECT * FROM nulls ORDER BY num DESC NULLS FIRST;
```

Обратите внимание на ключевое слово Backward в плане выполнения запроса. Что оно означает?

**Задание 2.** Модифицируйте команду создания индекса таким образом, чтобы он использовался при выполнении следующей выборки:

```
SELECT * FROM nulls ORDER BY num NULLS FIRST;
```

**Задание 3.** Выполните аналогичные эксперименты, задавая убывающий порядок сортировки с помощью ключевого слова DESC и изменяя расположение значений NULL в выборке с помощью ключевых слов NULLS FIRST и NULLS LAST предложения ORDER BY. С помощью команды EXPLAIN ANALYZE посмотрите, каким будет фактическое время выполнения команд. За дополнительной информацией обратитесь к описанию команды CREATE INDEX, приведенному в документации.

15. Обратитесь к запросам в главе 6. Выполните команду EXPLAIN для всех этих запросов и ознакомьтесь с планами, которые создаст планировщик. В планах могут встречаться наименования методов, которые не были рассмотрены в тексте главы, однако они должны быть вам интуитивно понятны.

16. В разделе документации 19.7 «Планирование запросов» приведены параметры, с помощью которых можно влиять на решения, принимаемые планировщиком. В тексте главы мы уже говорили о параметрах, управляющих выбором способа соединения наборов строк, и показали простой пример. Также было сказано и о том, что при установке значений параметров enable_hashjoin, enable_mergejoin и enable_nestloop в «off» не накладывается полного запрета на использование соответствующих методов. Вместо этого конкретному методу назначается очень высокая стоимость. Давайте проведем следующий эксперимент: запретим использование всех методов соединения наборов строк и выполним запрос, в котором соединяются две таблицы.

```
SET enable_hashjoin = off;
```

```
SET
```

```
SET enable_mergejoin = off;
```

```
SET
```

```
SET enable_nestloop = off;
```

```
SET
```

Запрос выводит информацию о числе мест в самолетах всех моделей.

```
EXPLAIN
 SELECT a.model, count( * )
  FROM aircrafts a, seats s
  WHERE a.aircraft_code = s.aircraft_code
  GROUP BY a.aircraft_code;
```

```
                              QUERY PLAN
----------------------------------------------------------------------
 GroupAggregate
 (cost=10000000000.41..10000000109.95 rows=9 width=56)
   Group Key: a.aircraft_code
   ->  Nested Loop
       (cost=10000000000.41..10000000103.16 rows=1339 width=48)
         ->  Index Scan using aircrafts_pkey on aircrafts a
             (cost=0.14..12.27 rows=9 width=48)
         ->  Index Only Scan using seats_pkey on seats s
```

```
                (cost=0.28..8.61 rows=149 width=4)
                  Index Cond: (aircraft_code = a.aircraft_code)
(6 строк)
```

Обратите внимание на оценки стоимости выполнения запроса. Резкое повышение оценок происходит именно в узле, отвечающем за соединение наборов строк. Эти оценки не означают, что время выполнения запроса будет стремиться к бесконечности. С помощью команды EXPLAIN ANALYZE выполните запрос и убедитесь в этом сами.

**Задание.** Самостоятельно ознакомьтесь с содержанием раздела документации 19.7 «Планирование запросов», а также раздела 14.3 «Управление планировщиком с помощью явных предложений JOIN» и проведите эксперименты с запросами, приведенными в главе 6 пособия, получая различные варианты планов и сравнивая их.

Ваша задача — понять, как изменения значений этих параметров влияют на план выполнения запроса. Однако для того чтобы понимать, *когда и почему* нужно изменять значения конкретных параметров, правильно оценивать степень и направленность их влияния, понимать взаимосвязь параметров, требуется опыт и изучение документации.

17. Самостоятельно ознакомьтесь с разделом документации 14.2 «Статистика, используемая планировщиком».

18. Команда EXPLAIN имеет опцию BUFFERS. Ознакомьтесь с ней самостоятельно по разделу документации 14.1 «Использование EXPLAIN».

19. При массовом вводе данных в базу данных производительность СУБД может снижаться по ряду причин, например, при наличии индексов они обновляются при вводе каждой новой строки в таблицу, а это требует дополнительных затрат ресурсов. Для повышения производительности СУБД в подобных ситуациях в документации предлагается ряд мер, например, удаление индексов перед началом массового ввода данных и пересоздание индексов после завершения такого ввода. Ознакомьтесь с этими мерами самостоятельно по разделу документации 14.4 «Наполнение базы данных». Смоделируйте ситуации, описанные в этом разделе документации, и выполните рекомендуемые действия.

# 11 Рекомендуемые источники

[1] Гарсиа-Молина, Г. Системы баз данных. Полный курс: пер. с англ. / Гектор Гарсиа-Молина, Джеффри Д. Ульман, Дженнифер Уидом. — М.: Вильямс, 2003. — 1088 с.: ил.

[2] Грофф, Дж. Р. SQL. Полное руководство: пер. с англ. / Джеймс Р. Грофф, Пол Н. Вайнберг, Эндрю Дж. Оппель. — 3-е изд. — М.: Вильямс, 2015. — 960 с.: ил.

[3] Дейт, К. Дж. Введение в системы баз данных: пер. с англ. / Крис Дж. Дейт. — 8-е изд. — М.: Вильямс, 2005. — 1328 с.: ил.

[4] Новиков Б. Настройка приложений баз данных / Борис Новиков, Генриетта Домбровская. — СПб.: БХВ-Петербург, 2012. — 240 с.: ил.

[5] Селко, Д. Стиль программирования Джо Селко на SQL: пер. с англ. / Джо Селко. — М.: Русская редакция; СПб.: Питер, 2006. — 206 с.: ил.

[6] Официальный сайт PostgreSQL: `http://www.postgresql.org`

[7] Postgres Professional: `http://postgrespro.ru`

*Учебное издание*

Моргунов Евгений Павлович
**ЯЗЫК SQL. БАЗОВЫЙ КУРС**

*Учебно-практическое пособие*

при поддержке Postgres Professional
`http://postgrespro.ru`

Редакторы *Е. В. Рогов, П. В. Лузанов*
Оригинал-макет и верстка *И. Е. Панченко*
Обложка *А. В. Климковский*

256