

Компания Postgres Professional

Е. П. Моргунов

PostgreSQL. Основы языка SQL

Учебное пособие

Санкт-Петербург

«БХВ-Петербург»

2018

УДК 004.655

ББК 32.973.26-018.2

М79

**Моргунов, Е. П.**

М79 PostgreSQL. Основы языка SQL: учеб. пособие / Е. П. Моргунов; под ред.

* В. Рогова, П. В. Лузанова. — СПб.: БХВ-Петербург, 2018. — 336 с.: ил. ISBN 978-5-9775-4022-3

Учебно-практическое пособие охватывает первую, базовую, часть учеб-ного курса по языку SQL, созданного при участии российской компании Postgres Professional. Учебный материал излагается в расчете на использо-вание системы управления базами данных PostgreSQL. Рассмотрено созда-ние рабочей среды, описаны языки определения данных и основные опе-рации выборки и изменения данных. Показаны примеры использования транзакций, уделено внимание методам оптимизации запросов. Матери-ал сопровождается многочисленными практическими примерами. Посо-бие может использоваться как для самостоятельного обучения, так и при проведении занятий под руководством преподавателя.

*Для программистов и студентов*

УДК 004.655

ББК 32.973.26-018.2

Рецензент: *Б. А. Новиков*, д-р. физ.-мат. наук, профессор СПбГУ

ISBN 978-5-9775-4022-3

ISBN 978-5-6041193-2-7 © ООО «ППГ», 2018

**Оглавление**

**Предисловие автора** **5**

**Введение** **7**

**Глава 1. Введение в базы данных и SQL** **13**

1.1. Что такое базы данных и зачем они нужны . . . . . . . . . . . . . . . . . . 13

1.2. Основные понятия реляционной модели . . . . . . . . . . . . . . . . . . . 15

1.3.ЧтотакоеязыкSQL................................ 18

1.4. Описание предметной области и учебной базы данных . . . . . . . . . . 19

Контрольныевопросыизадания........................... 23

**Глава 2. Создание рабочей среды** **25**

2.1.УстановкаСУБД.................................. 25

2.2. Программа psql — интерактивный терминал PostgreSQL . . . . . . . . . . 26

2.3. Развертывание учебной базы данных . . . . . . . . . . . . . . . . . . . . . 27

Контрольныевопросыизадания........................... 29

**Глава 3. Основные операции с таблицами** **31**

Контрольныевопросыизадания........................... 48

**Глава 4. Типы данных СУБД PostgreSQL** **51**

4.1.Числовыетипы .................................. 51

4.2. Символьные(строковые)типы . . . . . . . . . . . . . . . . . . . . . . . . . 54

4.3.Типы«дата/время»................................ 56

4.4.Логическийтип.................................. 63

4.5.Массивы ...................................... 64

4.6.ТипыJSON..................................... 68

Контрольныевопросыизадания........................... 73

**Глава 5. Основы языка определения данных** **95**

5.1. Значения по умолчанию и ограничения целостности . . . . . . . . . . . . 95

5.2. Созданиеиудалениетаблиц ..........................105

5.3.Модификациятаблиц ..............................117

5.4.Представления ..................................123

5.5.Схемыбазыданных................................131

Контрольныевопросыизадания...........................133

**Глава 6. Запросы** **145**

6.1. Дополнительные возможности команды SELECT . . . . . . . . . . . . . . 145

6.2.Соединения ....................................152

3

*Оглавление*

6.3. Агрегированиеигруппировка . . . . . . . . . . . . . . . . . . . . . . . . . 168

6.4.Подзапросы ....................................176

Контрольныевопросыизадания...........................192

**Глава 7. Изменение данных** **211**

7.1. Вставкастроквтаблицы.............................211

7.2. Обновлениестроквтаблицах..........................219

7.3. Удалениестрокизтаблиц............................224

Контрольныевопросыизадания...........................226

**Глава 8. Индексы** **241**

8.1.Общаяинформация ...............................241

8.2. Индексыпонесколькимстолбцам . . . . . . . . . . . . . . . . . . . . . . . 246

8.3.Уникальныеиндексы...............................247

8.4. Индексынаосновевыражений . . . . . . . . . . . . . . . . . . . . . . . . . 248

8.5.Частичныеиндексы................................248

Контрольныевопросыизадания...........................250

**Глава 9. Транзакции** **255**

9.1.Общаяинформация ...............................255

9.2. Уровень изоляции Read Uncommitted . . . . . . . . . . . . . . . . . . . . . 259

9.3. Уровень изоляции Read Committed . . . . . . . . . . . . . . . . . . . . . . . 261

9.4. Уровень изоляции Repeatable Read . . . . . . . . . . . . . . . . . . . . . . . 265

9.5. Уровень изоляции Serializable . . . . . . . . . . . . . . . . . . . . . . . . . . 269

9.6. Пример использования транзакций . . . . . . . . . . . . . . . . . . . . . . 275

9.7. Блокировки ....................................278

Контрольныевопросыизадания...........................280

**Глава 10. Повышение производительности** **293**

10.1.Основныепонятия ................................293

10.2.Методыпросмотратаблиц............................296

10.3. Методы формирования соединений наборов строк . . . . . . . . . . . . . 302

10.4.Управлениепланировщиком . . . . . . . . . . . . . . . . . . . . . . . . . . 305

10.5.Оптимизациязапросов .............................311

Контрольныевопросыизадания...........................317

**Рекомендуемые источники** **328**

**Предметный указатель** **329**

4

**Предисловие автора**

* системой управления базами данных PostgreSQL мне довелось познакомиться в 1998 году, это была еще версия 6.2. Важную роль в моей встрече с PostgreSQL сыг-рал Вадим Михеев, который в те годы был одним из ключевых разработчиков этой СУБД. После перехода в 2000 году на преподавательскую работу в вуз мне потребова-лась свободно распространяемая СУБД для использования в учебном процессе. На эту роль идеально подходила СУБД PostgreSQL.

Книга, которую вы держите в руках, появилась по инициативе и при поддержке компании Postgres Professional — российского поставщика СУБД PostgreSQL. Летом 2016 года я встретился с директором компании Олегом Бартуновым. Он высказал идею о том, что для распространения СУБД PostgreSQL необходим учебник по язы-ку SQL. PostgreSQL является самой продвинутой свободно распространяемой СУБД с очень хорошей поддержкой стандарта SQL. Тем не менее существуют и специфиче-ские черты, присущие именно PostgreSQL, поэтому такой учебник должен был отра-жать эту специфику. Поскольку я уже много лет использовал PostgreSQL в преподава-нии в вузе, то Олег Бартунов предложил мне заняться написанием такого учебника. Это предложение показалось мне очень интересным.

Мы обсудили план будущего учебника с Павлом Лузановым и Егором Роговым, кото-рые занимаются образовательными программами компании. Они согласились вы-ступить в трудной и ответственной роли редакторов книги. Координировали работу над учебником и помогали на разных стадиях этого трудного процесса Олег Бартунов

* заместитель директора компании Иван Панченко. Оригинальную обложку книги сделал Андрей Климковский, а Егор Рогов подготовил предметный указатель, благо-даря которому книгой будет удобно пользоваться. Вообще, без поддержки компании Postgres Professional учебник в таком виде не смог бы состояться.

Данный учебник рекомендуется использовать в комплексе с учебником по основам технологий баз данных, который написан профессором Санкт-Петербургского госу-дарственного университета Борисом Асеновичем Новиковым также при поддержке компании Postgres Professional. Важным является то, что в обоих учебниках для ил-люстрации приемов применения языка SQL используется одна и та же учебная база данных «Авиаперевозки», разработанная Павлом Лузановым и Егором Роговым. Она содержит данные, по своим свойствам близкие к реальным. Это позволяет в каче-стве упражнений моделировать ситуации, которые могли бы иметь место в реальной работе специалиста по базам данных.

5

*Предисловие автора*

Знанием только языка SQL квалификация специалиста по базам данных не исчер-пывается, поэтому на сайте компании Postgres Professional представлен целый ряд «фирменных» учебных курсов по PostgreSQL. Ссылка на них приведена в конце кни-ги в разделе «Рекомендуемые источники».

* планах автора и компании Postgres Professional — выпуск второй части учебни-ка. Она будет посвящена более сложным вопросам использования языка SQL в сре-де СУБД PostgreSQL. Предполагается рассмотреть такие темы, как хранимые функ-ции и процедуры, язык PL/pgSQL, триггеры, полнотекстовый поиск, расширяемость PostgreSQL.
  + *П. Моргунов*

**Введение**

* 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, больше-го числа специалистов, способных проектировать их и обслуживать. В ходе эволю-ции теории и практики баз данных стандартом де-факто стала реляционная модель данных, а в рамках этой модели сформировался и специализированный язык про-граммирования, позволяющий выполнять все необходимые операции с данными — Structured Query Language (SQL). Таким образом, важным компонентом квалифика-ции специалиста в области баз данных является владение языком SQL.
* настоящем учебном пособии излагаются основы языка SQL — это базовый курс. Причем язык рассматривается применительно к конкретной системе управления ба-зами данных (СУБД) — PostgreSQL. Реализация языка SQL в каждой СУБД соответ-ствует стандарту в той или иной степени, но кроме стандартизированных функций и возможностей, каждая СУБД предлагает и свои дополнительные расширения язы-ка. 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, на-пример, что пользователь (конечно, являющийся специалистом в области баз дан-ных) может разработать свои собственные типы данных. Эти типы данных будут об-ладать 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 плат-форм.
* пособии рассматриваются не только все основные команды языка SQL, но также и другие вопросы, такие как индексы и транзакции.

Книга написана таким образом, чтобы ее можно было использовать как под руковод-ством преподавателя, так и самостоятельно. Предполагается, что студенты имеют до-ступ к уже установленной СУБД, поэтому процедура установки PostgreSQL детально не рассматривается, а лишь даются указания о том, где найти инструкции по уста-новке.

Это пособие предназначено для получения практических навыков использования языка SQL. Учебный материал подается таким образом, что многие важные знания

7

*Введение*

читатель должен получить в результате выполнения заданий, находящихся в кон-це каждой главы. В основном тексте глав эти знания могут быть не представлены. Предполагается, что значительная часть заданий будет выполняться читателем са-мостоятельно с помощью документации на СУБД PostgreSQL, но зачастую даются и указания к их выполнению. Задания, приведенные в пособии, различаются по уров-ню 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 кур-сом, все же в первой главе кратко, на элементарном уровне излагаются основные по-нятия теории баз данных и реляционной модели. Это сделано для того, чтобы студен-ты 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 тра-диционно 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 изу-чение 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 система PostgreSQL уже должна быть установлена в учебной аудитории заранее.

Глава 3 представляет собой краткий обзор основных возможностей языка SQL, по-сле ее изучения студенты должны представлять себе простые способы использования

8

*Введение*

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 СУБД PostgreSQL. Это большая глава, однако в ней значительную часть составляют задания

* упражнения. Предполагается, что студенты за четыре часа должны усвоить только основные приемы использования типов данных. А для того, чтобы знания закрепи-лись, рекомендуется обращаться к материалу этой главы (в том числе и к упражнени-ям) в процессе изучения остальных глав пособия при необходимости уточнения тех или иных особенностей применения конкретных типов. Распределить время, выде-ленное на изучение этой главы, мы рекомендуем следующим образом: два часа на первые четыре раздела — числовые и строковые типы, типы «дата/время» и логиче-ский тип, еще два часа — на массивы и тип json/jsonb.

Чтобы выполнять запросы к базе данных, необходимо хорошо понимать ее структу-ру, взаимосвязи таблиц. Поэтому глава 5, в которой рассматриваются основы языка определения данных, очень важна с точки зрения детального изучения таблиц ба-зы 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 манипулирова-ния данными. Поэтому глава 5 «Основы языка определения данных» предшествует главе 6 «Запросы». Однако избранный подход реализуется не слишком жестко: в об-зорной главе 3 рассматриваются основные команды, в том числе и несложные за-просы. А запросы — это уже язык манипулирования данными. На изучение главы 5 отведено четыре часа. В течение первого двухчасового занятия нужно изучить два первых раздела, в которых освещаются такие вопросы, как ограничения целостно-сти и создание и удаление таблиц. Второе двухчасовое занятие нужно посвятить изу-чению трех оставшихся разделов. 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 разделов. Первый из них посвящен разнообразным дополнительным возможностям команды SELECT. Речь идет, в частности, о таких вещах, как предложения LIMIT

* OFFSET, оператор LIKE и регулярные выражения в условиях предложения WHERE
* о других возможностях. Тем не менее материал этого раздела несложный, для его изучения достаточно выделить один час. Во втором разделе рассказывается о спосо-бах соединения таблиц. Это более сложная тема, поэтому для ее изучения необходи-мо выделить два часа. Третий раздел посвящен агрегированию и группировке. В нем

9

*Введение*

рассматривается и такая важная и интересная тема, как оконные функции. Данный раздел также требует двухчасового занятия. Самый сложный раздел этой главы — четвертый. Он посвящен подзапросам. В нем, в частности, освещается такая важная

* интересная тема, как общие табличные выражения (Common Table Expressions — CTE). Для изучения материала данного раздела необходимо выделить три часа.
* главе 7 собраны все команды, предназначенные для изменения данных: вставка строк, их обновление и удаление. Поскольку в предшествующих главах эти команды уже применялись для решения простых задач, то в данной главе рассматриваются более сложные способы их использования. В ней много упражнений, они составля-ют 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 произво-дительности, т. е. скорости выполнения запросов, то было бы целесообразно после изучения заключительной главы вернуться к главе 8 и посмотреть на представлен-ные 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 объ-яснить полученные результаты.

* заключительной главе 10 рассматриваются вопросы повышения производительно-сти. Эта глава может показаться слишком абстрактной и сложной для начального кур-са языка SQL, тем не менее она очень важна. Студенты должны научиться читать пла-ны выполнения запросов и понимать назначение каждой операции, представленной в плане. А овладение искусством оптимизации запросов потребует много времени и опыта, оно придет не сразу.
* том случае, когда на практические занятия по дисциплине «Базы данных» в учеб-ном плане отводится 54 часа, можно изменить предлагаемое распределение учеб-ных часов. В частности, в главе 4 можно больше времени посвятить типам данных json/jsonb и массивам. В главе 6 можно более детально рассмотреть оконные функ-ции и общие табличные выражения. При изучении главы 9, посвященной транзакци-ям, было бы целесообразно разработать несложное приложение, в котором исполь-зовались бы транзакции, и провести эксперименты с этим приложением, выполняя параллельно несколько сеансов и изменяя при этом уровни изоляции транзакций.
* рамках главы 10 имеет смысл вернуться к командам и запросам главы 8 и изучить

10

*Введение*

планы их выполнения с помощью команды EXPLAIN. За счет дополнительного вре-мени можно рассмотреть все задания и упражнения повышенной сложности (поме-ченные 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 |

* пособии используются различные виды шрифтов для выделения фрагментов тек-ста в зависимости от их назначения. Команды, вводимые пользователем как в среде операционной системы, так и в среде утилиты psql, выделяются полужирным моно-ширинным шрифтом. Например:

**psql -d demo -U postgres**

или

**SELECT avg( total\_amount ) FROM bookings;**

Результаты работы команд операционной системы и SQL-команд, выполняемых

* 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 спо-собствовать повышению уровня вашей квалификации и расширению профессио-нального кругозора.

11

**Глава 1**

**Введение в базы данных и SQL**

Эта глава — вводная. В ней мы расскажем об основах баз данных, о том, что такое реляционная мо-дель 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 струк-туру: данные хранились в виде записей, разделенных на поля фиксированной длины.

* 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**. Мож-но 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 дан-ных, которая сейчас и является доминирующей. Поэтому в настоящее время преоб-ладают базы данных реляционного типа. Их характерной чертой является тот факт, что данные воспринимаются пользователем как таблицы. В распоряжении пользова-теля 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 поддержи-вать связи между элементами данных, избавляя программиста от необходимости за-ниматься этой рутинной и очень трудоемкой работой. В те времена, когда технологии

13

*Глава 1. Введение в базы данных и SQL*

реляционных баз данных еще не получили широкого распространения, программи-стам приходилось на процедурных языках вручную реализовывать такие операции, которые сейчас называются каскадным обновлением внешних ключей или каскад-ным 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 удаляе-мой записи. Тот подход к работе с базами данных назывался навигационным — про-граммист 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 экза-менационные оценки — в другом файле, который условно назовем «Успеваемость». 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 «Успеваемость», 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 программирова-ния на «атомарном» уровне, потому что современные языки для «общения» с этими базами данных являются декларативными. Это означает, что для получения резуль-тата 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 си-стема включает в себя программное и аппаратное обеспечение, сами данные, а также пользователей.

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

Основным программным обеспечением является система управления базами дан-ных. По-английски она называется database management system (DBMS). Кроме СУБД

* 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 ка-тегория — это прикладные программисты. Вторая категория — это конечные поль-зователи, ради которых и выполняется вся работа. Они могут получить доступ к базе

14

*1.2. Основные понятия реляционной модели*

данных, используя прикладные программы или универсальные приложения, кото-рые 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 пользо-вателей — это администраторы базы данных. В их обязанности входят: создание ба-зы данных, выбор оптимальных режимов доступа к ней, разграничение полномочий различных пользователей на доступ к той или иной информации в базе данных, вы-полнение 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 являют-ся различные приложения, написанные прикладными программистами, или встро-енные 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 выпол-нять свои функции.

**1.2. Основные понятия реляционной модели**

* каждой технологической сфере есть своя терминология. Существуют базовые тер-мины, на которых основываются все дальнейшие рассуждения. Такие термины при-сутствуют и в сфере баз данных. Сейчас мы кратко о них поговорим.
* эпоху, предшествующую рождению реляционной теории, базы данных традици-онно рассматривались как набор **файлов**, состоящих из **записей**, а записи, в свою очередь, подразделялись на отдельные **поля**. Поле являлось элементарной единицей данных.
* реляционных базах данных пользователь воспринимает данные в виде таблиц. По-этому 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 пересечениикоторых должны находиться «атомарные» значения, которые нельзя разбить на бо-лее мелкие элементы без потери смысла.
* формальной теории реляционных баз данных эти таблицы называют **отношения-ми (relations)** — поэтому и базы данных называются реляционными. Отношение —

15

*Глава 1. Введение в базы данных и SQL*

это математический термин. При определении свойств таких отношений использу-ется теория множеств. В терминах данной теории строки таблицы будут называться **кортежами (tuples)**, а колонки — **атрибутами**. Отношение имеет заголовок, кото-рый состоит из атрибутов, и тело, состоящее из кортежей. Количество атрибутов на-зывается **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 терми-нов. 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 **ограничени-ям**, 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 учиты-вать, например, год поступления студента в вуз);

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

16

*1.2. Основные понятия реляционной модели*

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

– оценка может принимать только три значения — 3 (удовлетворительно), 4 (хоро-шо) 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 табли-цы, позволяющая уникальным образом идентифицировать строки в ней. Ключ может состоять 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, удо-стоверяющего 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 *неизбыточным*, т. е. никакое подмножество атрибутов, входящих в него, не должно обладать свойством уникальности. Потенциальный ключ, включающий два упомянутых атрибута, явля-ется неизбыточным.

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

* таким номером зачетной книжки в таблицу «Успеваемость» не имеет смысла. Та-ким 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 то-го, что называется **внешним ключом**. Таблица, содержащая внешний ключ, назы-вается **ссылающейся** таблицей (referencing table). Таблица, содержащая соответству-ющий 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 обес-печения 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 клю-чей, называется **ограничением ссылочной целостности (ссылочным ограниче-нием)**.

17

*Глава 1. Введение в базы данных и SQL*

Обеспечением выполнения ограничений ссылочной целостности занимается СУБД,

* от разработчика требуется лишь указать атрибуты, служащие в качестве внешних ключей. При проектировании баз данных часто предусматривается, что при удале-нии строки из ссылочной таблицы соответствующие строки из ссылающейся таблицы должны быть также удалены, а при изменении значения столбца, на который ссыла-ется внешний ключ, должны быть изменены значения внешнего ключа в ссылающей-ся таблице. Этот подход называется **каскадным удалением (обновлением)**.

Иногда применяются и другие подходы. Например, вместо удаления строк из ссыла-ющейся таблицы в этих строках просто заменяют значения атрибутов, входящих во внешний ключ, так называемыми NULL-значениями. Это специальные значе-ния, 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 ра-боты, 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 уда-ление связанных по внешнему ключу строк из таблицы «Успеваемость».

**1.3. Что такое язык SQL**

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

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

18

*1.4. Описание предметной области и учебной базы данных*

* операторам языка манипулирования данными (DML) относятся команды для вы-борки 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 началь-ном 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 потребо-вало слишком много времени. Но вместе с тем она должна быть достаточно разнооб-разной, чтобы запросы к ней выглядели бы правдоподобными, почти такими же, как и в реальной работе.

* качестве предметной области выберем пассажирские авиаперевозки. Ее ориги-нальное описание и описание базы данных «Авиаперевозки» можно найти по адре-сам <https://postgrespro.ru/education/demodb>и [https://postgrespro.ru/docs/postgrespro/](https://postgrespro.ru/docs/postgrespro/current/demodb-bookings.html) [current/demodb-bookings.html.](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 име-ет определенный код, который присваивает Международная ассоциация авиапере-возчиков (IATA). При этом будем считать, что самолеты одной модели имеют оди-наковые компоновки салонов, т. е. порядок размещения кресел и нумерацию мест

* 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 аэро-порту присвоен уникальный трехбуквенный код, при этом используются только за-главные буквы латинского алфавита. Эти коды присваивает не сама авиакомпания,

* специальные организации, управляющие пассажирскими авиаперевозками. Зача-стую название аэропорта не совпадает с названием того города, которому этот аэро-порт принадлежит. Например, в городе Новосибирске аэропорт называется Толмаче-во, в городе Екатеринбурге — Кольцово, а в Санкт-Петербурге — Пулково. К тому же

19

*Глава 1. Введение в базы данных и SQL*

некоторые города имеют более одного аэропорта. Сразу в качестве примера вспоми-нается Москва с ее аэропортами Домодедово, Шереметьево и Внуково. Добавим еще одну важную деталь: каждый аэропорт характеризуется географическими координа-тами — долготой и широтой, а также часовым поясом.

Формируются маршруты перелетов между городами. Конечно, каждый такой марш-рут требует указания не только города, но и аэропорта, поскольку, как мы уже сказа-ли, 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 ше-стизначный номер, включающий цифры и буквы латинского алфавита.

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

При фактическом выполнении рейса возникает необходимость в учете дополнитель-ных сведений, а именно: фактического времени отправления и фактического време-ни 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 на-стоящее время общепринятой практикой является оформление электронных биле-тов. Каждый такой билет имеет уникальный номер, состоящий из 13 цифр. В рамках одной процедуры бронирования может быть оформлено несколько билетов, но каж-дая 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.

* каждый билет, кроме его тринадцатизначного номера, записывается идентифика-тор пассажира, а также его имя и фамилия (в латинской транскрипции) и контактные данные. В качестве идентификатора пассажира используется номер документа, удо-стоверяющего личность. Конечно, пассажир может сменить свой документ, а иной

20

*1.4. Описание предметной области и учебной базы данных*

раз даже фамилию и имя, за время, прошедшее между бронированием билетов в раз-ные дни, поэтому невозможно наверняка сказать, что какие-то конкретные билеты были оформлены на одного и того же пассажира.

* каждый электронный билет может быть вписано более одного перелета. Специа-листы называют эти записи о перелетах сегментами. В качестве примера наличия нескольких сегментов можно привести такой: Красноярск — Москва, Москва — Ана-па, Анапа — Москва, Москва — Красноярск. При этом возможно в рамках одного бронирования оформить несколько билетов на различных пассажиров. Для каждо-го 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:

* талоне указывается такой же номер, который имеет электронный авиабилет дан-ного пассажира. Кроме того, в талоне указывается номер рейса и номер места в са-молете. 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 оформ-лении посадочных талонов. Если, например, пассажир приобрел билет с экономиче-ским 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 мо-дельную 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,

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

21

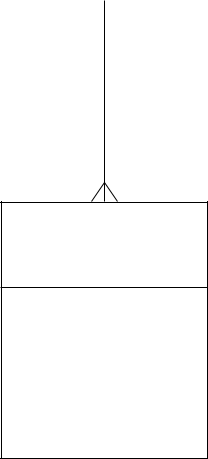
*Глава 1. Введение в базы данных и SQL*

**Bookings**

Бронирования

# book\_ref

book\_date total\_amount



**Tickets**

Билеты

# ticket\_no

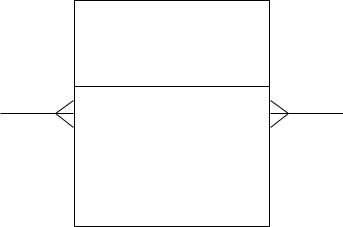
book\_ref

passenger\_id

passenger\_name

contact\_data

**Ticket\_flights**

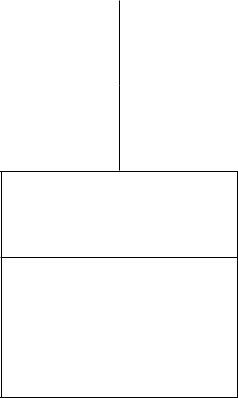


Перелеты

# ticket\_no

# flight\_id

fare\_conditions amount



**Boarding\_passes**

Посадочные талоны

* ticket\_no
* flight\_id

boarding\_no seat\_no

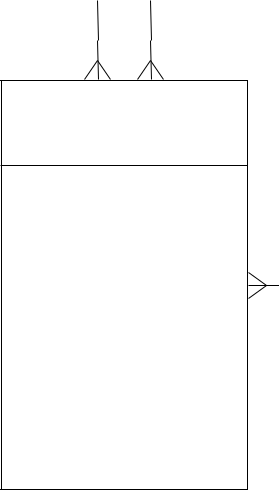
**Airports**

Аэропорты

# airport\_code

airport\_name city

longitude latitude timezone



**Flights**

Рейсы

# flight\_id

flight\_no

scheduled\_departure scheduled\_arrival

departure\_airport arrival\_airport

status

aircraft\_code

actual\_departure actual\_arrival

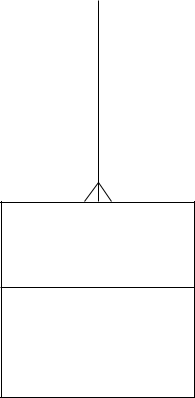
**Aircrafts**

Самолеты

# aircraft\_code



model range



**Seats**

Места

* aircraft\_code
* seat\_no

fare\_conditions

22

*Контрольные вопросы и задания*

**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 реализации базы данных «Авиаперевозки» предполагается, что самолеты од-ной модели могут иметь только одну компоновку салона. Представим, что руко-водством принято решение о том, что нужно учитывать возможность наличия различных компоновок для каждой модели. Какие таблицы придется модифи-цировать в таком случае и каким образом? Потребуется ли создавать дополни-тельные таблицы?

23

**Глава 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 удален-ному 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.

* 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 ука-заниями 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 предлагает-ся компанией Postgres Professional. Он называется Postgres Pro и содержит не только все функции и модули, входящие в состав стандартного дистрибутива, но и дополни-тельные разработки, выполненные в компании Postgres Professional. Для изучения основ языка SQL эти дистрибутивы подходят в равной степени. Однако документа-ция 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 най-ти инструкции по его установке в различных операционных системах можно по ад-ресу [https://www.postgresql.org/download/.](https://www.postgresql.org/download/)

25

*Глава 2. Создание рабочей среды*

Если же вы остановили свой выбор на дистрибутиве Postgres Pro, тогда следует обра-титься сюда: [https://postgrespro.ru/products/postgrespro/download/latest.](https://postgrespro.ru/products/postgrespro/download/latest)

После установки как PostgreSQL, так и Postgres Pro, в среде Windows придется пред-принять дополнительные меры, чтобы использование русского алфавита в интерак-тивном терминале psql не вызывало проблем. Утилита psql рассматривается в следу-ющем разделе.

* процессе установки будет создана отдельная 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 сер-вер баз данных, потому что иначе невозможно работать с данными. Как это сде-лать, подробно описано в документации в разделе 18.3 «Запуск сервера баз данных». Найти этот раздел можно по адресу [https://postgrespro.ru/docs/postgresql/current/](https://postgrespro.ru/docs/postgresql/current/server-start.html) [server-start.html.](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 сер-вера». Найти этот раздел можно по адресу [https://postgrespro.ru/docs/postgresql/9.6/](https://postgrespro.ru/docs/postgresql/9.6/server-shutdown.html) [server-shutdown.html.](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 тер-минал 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**

* среде утилиты psql можно вводить не только команды языка SQL, но и различные сервисные команды, поддерживаемые самой утилитой.

26

*2.3. Развертывание учебной базы данных*

Для получения краткой справки по всем сервисным командам нужно ввести

**\?**

Многие такие команды начинаются с символов «nd». Например, для того чтобы про-смотреть 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 кон-кретной 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 вводи-мой 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». Анало-гично можно поступить и со вторым словом: для его ввода достаточно ввести лишь буквы «ta» и нажать клавишу <Tab>. Если вы ввели слишком мало букв для того, что-бы утилита psql могла однозначно идентифицировать ключевое слово, дополнения не произойдет. Но в таком случае вы можете нажать клавишу <Tab> дважды и полу-чить 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 рас-смотрению вопроса о том, как развернуть в вашем кластере PostgreSQL учебную базу данных «Авиаперевозки», подготовленную компанией Postgres Professional.

27

*Глава 2. Создание рабочей среды*

На сайте компании есть раздел, посвященный этой базе данных, найти его можно по ссылке [https://postgrespro.ru/education/demodb.](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,

* самая полная версия включает данные за целый год. Все данные были сгенерирова-ны с помощью специальных алгоритмов, обеспечивающих их «правдоподобность». Мы рекомендуем вам начать с компактной версии базы данных «Авиаперевозки»,
* 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 досту-па к данным.
* качестве первого шага к развертыванию базы данных нужно скачать ее заархивиро-ванную резервную копию по ссылке [https://edu.postgrespro.ru/demo\_small.zip.](https://edu.postgrespro.ru/demo_small.zip) Затем необходимо извлечь файл из архива:

**unzip demo\_small.zip**

Извлеченный файл называется demo\_small.sql. Теперь создадим базу данных с име-нем demo в вашем кластере PostgreSQL. Самый краткий вариант команды будет та-ким:

**psql -f demo\_small.sql -U postgres**

Если вы хотите перенаправить вывод сообщений, которые генерирует СУБД в про-цессе 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.

28

*Контрольные вопросы и задания*

Если бы наш SQL-файл был очень большим, тогда можно было бы выполнить коман-ду 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 дан-ных 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 опе-рационной системы.
2. Ознакомьтесь с утилитой psql с помощью встроенной справки, а также с помо-щью справки, вызываемой по команде

**psql --help**

1. Кроме утилиты psql существуют и другие универсальные программы для рабо-ты с сервером баз данных PostgreSQL, например, pgAdmin. Это мощная утилита с графическим интерфейсом.

Самостоятельно установите программу pgAdmin и изучите основные приемы работы с ней.

1. Разверните учебную базу данных. Попробуйте подключиться к ней с помощью утилиты psql. Для выхода из утилиты используйте команду nq.

29

**Глава 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 основ-ных 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.

* 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 данной главе нашего пособия вы ознакомитесь с основными командами языка SQL, которые позволят вам выполнять базовые операции. А более сложные (и интересные) команды вы изучите в следую-щих главах.

Скажем два слова о нашем подходе к работе. В принципе возможны два способа орга-низации 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 ква-лифицированным 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, по-скольку он требует меньше усилий на начальном этапе освоения языка SQL.

Однако, на наш взгляд, более правильным является другой способ. Наверное, он бо-лее трудоемкий, но при его использовании вы лучше, как говорится, прочувствуете процесс создания таблиц и ввода записей в эти таблицы. А при выполнении раз-личных запросов 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.

31

*Глава 3. Основные операции с таблицами*

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 вруч-ную — нерационально. Гораздо более рациональным будет их автоматическое фор-мирование программным путем.

* главе 1 мы описали предметную область, поэтому сейчас можем приступить к непо-средственному 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 синтак-сис 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]**

**);**

* 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**

32

*Глава 3. Основные операции с таблицами*

* 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 | range | Числовой | integer | NOT NULL |
| дальность полета, км |  |  |  | range > 0 |

Типы char и text являются символьными типами данных и позволяют вводить лю-бые 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 количе-ства разных символов, поэтому для атрибута «Модель самолета» мы выбрали тип данных text, который не требует указания максимальной длины сохраняемого зна-чения. Вообще, число символов, которые можно сохранить в поле типа text, прак-тически не ограничено.

Для атрибута «Максимальная дальность полета» мы выбрали целый числовой тип. Значения всех атрибутов каждой строки данной таблицы должны быть определен-ными, поэтому на них накладывается ограничение NOT NULL. В принципе в таблицах базы данных могут содержаться неопределенные значения некоторых атрибутов. Го-воря 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.

* 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 ис-пользуется на практике. В отличие от естественных ключей иногда используются и так называемые **суррогатные ключи**, но о них мы расскажем в последующих главах пособия.

33

*Глава 3. Основные операции с таблицами*

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 ко-мандной 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 традицион-но ключевые слова языка SQL вводят в верхнем регистре, что повышает наглядность SQL-операторов. Тем не менее наименования типов данных (integer, char, text

* 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 выпол-нении 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 вы-полняемого 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 коман-да 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 напе-чатана 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 при-глашения к вводу команд, выводимого утилитой psql, будет отличаться от первона-чального. В конце команды необходимо поставить точку с запятой.

34

*Глава 3. Основные операции с таблицами*

**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(# );**

* 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 <"> вызвать на экран (из буфера истории введенных команд) всю команду полностью в компактном виде и при необходимо-сти отредактировать ее либо выполнить еще раз без редактирования. При этом для команды, введенной построчно, сохраняется ее построчная структура, а приглаше-ние 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 редактирова-нии нужно использовать клавиши < > и <!>, но не <"> или <#>.

Если вы хотите непосредственно из среды psql вызвать внешний редактор для редак-тирования текущего буфера запроса, то нужно воспользоваться командой ne.

Если вы решили прервать ввод команды, еще не введя ее полностью, то просто на-жмите клавиши <Ctrl>+<C>, в результате ввод команды будет прерван, а приглаше-ние к вводу, выводимое утилитой psql, примет свой первоначальный вид:

**demo=# CREATE TABLE aircrafts**

* **aircraft\_code char( 3 ) NOT NULL, demo(# ^C**

**demo=#**

35

*Глава 3. Основные операции с таблицами*

Теперь выберите способ ввода команды для создания таблицы aircrafts и введите

* 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 ути-литы 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)

* этом выводе новым для вас может быть выражение public.aircrafts. В нем сло-во public означает имя так называемой **схемы**. Это, упрощенно говоря, раздел базы данных, в котором и создаются таблицы и другие объекты. По умолчанию исполь-зуется схема public. О схемах мы будем говорить более подробно в последующих главах пособия.
* 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 клю-чевых 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 ключе-вых 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 таб-лицы можно задать свои собственные имена для всех ограничений, однако делать это не обязательно. Мы не задавали никакого имени для ограничения, наложенного на

36

*Глава 3. Основные операции с таблицами*

атрибут range, поэтому ядро PostgreSQL также сгенерировало это имя автоматиче-ски — aircrafts\_range\_check.

Следует различать команды языка SQL и команды утилиты psql. Команды, начина-ющиеся с символа «n», 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 табли-цу 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 команд-ной для удаления таблицы.

**DROP TABLE имя-таблицы;**

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

**INSERT INTO имя-таблицы [( имя-атрибута, имя-атрибута, ... )] VALUES ( значение-атрибута, значение-атрибута, ... );**

* 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 нали-чие квадратных скобок. Они указывают, что список атрибутов в команде не является обязательным, но при вводе команды квадратные скобки вводить не нужно. Одна-ко если вы не привели список атрибутов, тогда вы обязаны в предложении VALUES задавать значения атрибутов с учетом того порядка, в котором они следуют в опре-делении 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 из-менения порядка существующих столбцов) вам придется корректировать и команду INSERT в ваших прикладных программах.

Давайте добавим одну строку в таблицу aircrafts. Обратите внимание на одинар-ные 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 );**

37

*Глава 3. Основные операции с таблицами*

В ответ мы получим сообщение об успешном добавлении этой строки:

INSERT 0 1

* этом сообщении числа 0 и 1 имеют конкретный смысл. Второе из них, т. е. 1, озна-чает количество добавленных строк — в данном случае была добавлена всего одна строка. А первое число 0 имеет отношение к внутреннему устройству PostgreSQL, ко-торое 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, упро-щенный 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;**

aircraft\_code | model | range

---------------+---------------------+-------

SU9 | Sukhoi SuperJet-100 | 3000

(1 строка)

Давайте добавим еще несколько строк в таблицу aircrafts. Команда INSERT поз-воляет сделать это за один раз. Вспомните о том, что можно редактировать ранее введенную команду, вызвав ее на экран при помощи клавиши <">. Как и при вво-де 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 );**

38

*Глава 3. Основные операции с таблицами*

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,

* котором строки выведены из нее по команде SELECT, то можно увидеть совпаде-ние этих порядков. Однако полагаться на такое совпадение нельзя, т. к. порядок мо-жет 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 допол-нительные 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 значе-ния — 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 |

39

*Глава 3. Основные операции с таблицами*

|  |  |  |  |
| --- | --- | --- | --- |
| 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. Давай-те выберем модели самолетов, у которых максимальная дальность полета находится в пределах от 4 до 6 тыс. 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 дан-ных 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 харак-теристики самолета Sukhoi SuperJet, и теперь дальность его полета стала на 500 км больше.

40

*Глава 3. Основные операции с таблицами*

**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»:

**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, уда-лим информацию о самолетах с дальностью полета более 10 000 км, а также с даль-ностью полета менее 3 000 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» нет ни одной строки. Для продолжения работы необхо-димо эти данные восстановить. Можно использовать несколько способов.

1. Ввести заново команды INSERT из текста пособия, которые вы уже вводили.
2. Используя клавиши <"> и <#>, найти в списке истории команд введенные ра-нее команды INSERT и повторно их выполнить.

41

*Глава 3. Основные операции с таблицами*

1. 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 утили-ты psql и выполнить.

* нашей учебной базе данных содержится несколько таблиц, связанных между собой. Для таблицы «Самолеты» (aircrafts) ближайшей «родственницей» является табли-ца «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 |
|  |  |  |  | Значения из списка: |
|  |  |  |  | Economy, Comfort, |
|  |  |  |  | Business |

Для атрибута «Код самолета, IATA» был выбран тип char(3), поскольку этот атрибут присутствует и в таблице «Самолеты».

Значения атрибута «Номер места» (seat\_no) состоят из числовой части, обозначаю-щей номер ряда кресел в салоне самолета, и латинской буквы, обозначающей пози-цию 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 симво-лы. 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 это-му списку будет обеспечиваться с помощью ограничения CHECK. Также выбираем тип данных varchar. Все допустимые значения имеют различные длины, но мы ориен-тируемся на самое длинное значение.

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

* качестве первичного ключа выбрана комбинация атрибутов «Код самолета, IATA» и «Номер места» — это составной ключ. Таким образом, первичный ключ будет **есте-ственным**. Как уже было сказано выше, это означает, что и в реальной предметной

42

*Глава 3. Основные операции с таблицами*

области существуют такие понятия, как код самолета и номер места, и эти понятия используются на практике.

* 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 табли-це «Самолеты» (aircrafts). Таблица «Места» называется ссылающейся (referencing), а таблица «Самолеты» — ссылочной (referenced).

Поскольку номера мест привязаны к модели самолета, то в случае удаления из таб-лицы «Самолеты» какой-либо строки с конкретным кодом самолета необходимо уда-лить также и из таблицы «Места» все строки, в которых значение атрибута «Код са-молета» такое 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 рас-пределении мест по классам обслуживания для этой модели.

Поэтому в предложении для определения внешнего ключа появляется важное допол-нение: 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. По-добные 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**

43

*Глава 3. Основные операции с таблицами*

|  |  |  |  |
| --- | --- | --- | --- |
| 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,

* тип данных varchar — character varying. Первичный ключ здесь составной — (aircraft\_code, seat\_no). Ограничение CHECK, накладываемое на значения атри-бута 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 рассмотренной выше таблицей «Самоле-ты» является наличие ограничения внешнего ключа. Это ограничение имеет имя seats\_aircraft\_code\_fkey, сгенерированное самой СУБД, поскольку мы не пред-ложили в команде CREATE TABLE никакого своего имени для этого ограничения, хотя, в принципе, имели право это сделать, если бы захотели.

Для просмотра 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)

44

*Глава 3. Основные операции с таблицами*

* первой колонке выведенной таблицы указана так называемая схема базы данных — public. Мы уже говорили, что схема — это обособленный до некоторой степени раз-дел базы данных. По умолчанию все объекты создаются в схеме public. В третьей колонке указан тип — «таблица». Кроме таблиц могут быть еще и представления.
* 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 123, то добавлять информацию

* 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 обслу-живания 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 по-вторите команду INSERT. Так придется поступить со всеми моделями самолетов.

Таблица «Места» заполнена необходимыми данными. Теперь решим еще одну зада-чу. Предположим, что нам нужно получить информацию о количестве мест в салонах

45

*Глава 3. Основные операции с таблицами*

для всех типов самолетов. Имея некоторый опыт в программировании на других язы-ках, 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 таки-ми 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 отдель-ные однотипные команды для всех моделей самолетов. Язык SQL позволяет упро-стить решение такой задачи за счет применения операции группирования строк на основе некоторого критерия. Этим критерием будет являться совпадение значе-ний атрибута «Код самолета» (aircraft\_code) у различных строк таблицы «Места» (seats).

* модифицированной команде вместо предложения WHERE будет добавлено предло-жение 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 |

1. | 140

(9 строк)

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

46

*Глава 3. Основные операции с таблицами*

**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 |

1. | 140
2. | 170

763 | 222

1. | 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

* экономический класс). В этом случае группировка выполняется уже по двум атри-бутам: aircraft\_code и fare\_conditions. Отсортируем выборку по тем же столб-цам, 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)

47

*Глава 3. Основные операции с таблицами*

**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 реструктуриза-ции 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 появилось сообщение. Если вы забыли структуру таблицы aircrafts, то можно вывести ее определение на экран с помощью команды

**\d aircrafts**

1. Предложение ORDER BY команды SELECT позволяет отсортировать данные при выводе. По умолчанию сортировка выполняется по возрастанию значений ат-рибута, указанного в этом предложении. Но можно упорядочить строки и по убыванию значения атрибута. Для этого нужно после имени атрибута в пред-ложении ORDER BY добавить ключевое слово DESC (это сокращение от слова descendant — убывающий порядок). Самостоятельно напишите команду для вы-борки всех строк из таблицы aircrafts, чтобы строки были упорядочены по убы-ванию значения атрибута «Максимальная дальность полета, км» (range).
2. 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, вслед-ствие чего дальность полета этого лайнера возросла ровно в два раза. Команда UPDATE позволяет увеличить значение атрибута range в строке, хранящей ин-формацию об этом самолете, даже не выполняя предварительно выборку с це-лью выяснения текущего значения этого атрибута. При присваивании нового

48

*Контрольные вопросы и задания*

значения атрибуту range можно справа от знака «=» написать не только чис-ловую 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 моде-ли — Sukhoi SuperJet, поэтому необходимо использовать условие WHERE. Затем

с помощью команды SELECT проверьте полученный результат.

1. Если в предложении WHERE команды DELETE вы укажете логически и синтак-сически корректное условие, но строк, удовлетворяющих этому условию, в таб-лице не окажется, то в ответ СУБД выведет сообщение

DELETE 0

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

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

49

**Глава 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 упо-требили 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.

* этой главе мы расскажем лишь о самых основных типах данных, поскольку в вашем распоряже-нии 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: цело-численные типы, числа фиксированной точности, типы данных с плавающей точкой, последовательные типы (serial).

* 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 различают-ся по количеству байтов, выделяемых для хранения данных. В PostgreSQL существу-ют псевдонимы для этих стандартизированных имен типов, а именно: int2, int4 и int8. Число байтов отражается в имени типа.

51

*Глава 4. Типы данных СУБД PostgreSQL*

При выборе конкретного целочисленного типа принимают во внимание диапазон допустимых значений и затраты памяти. Зачастую тип integer считается оптималь-ным 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 про-водить изложение на примере типа numeric. Для задания значения этого типа ис-пользуются 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. На-пример, у числа 12.3456 точность составляет 6 цифр, а масштаб — 4 цифры.

Параметры этого типа данных указываются в круглых скобках после имени типа:

numeric(точность, масштаб). Например, numeric(6, 2).

Его главное достоинство — это обеспечение *точных* результатов при выполнении вы-числений, когда это, конечно, возможно в принципе. Это оказывается возможным при выполнении сложения, вычитания и умножения. Числа типа numeric могут хра-нить очень большое количество цифр: 131 072 цифры — до десятичной точки (запя-той), 16 383 — после точки. Однако нужно учитывать, что такая точность достигается за счет замедления вычислений по сравнению с целочисленными типами и типами с плавающей точкой. 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 случа-ях, когда требуется гарантировать точность вычислений.

Представителями типов данных с плавающей точкой являются типы real и double precision. Они представляют собой реализацию стандарта IEEE «Standard 754 for Binary Floating-Point Arithmetic». Тип данных real может представить числа в диа-пазоне, 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 допу-стимой, то будет иметь место округление значения. А вот при вводе очень маленьких чисел, которые невозможно представить значениями, отличными от нуля, будет ге-нерироваться ошибка потери значимости, или исчезновения значащих разрядов (an underflow error).

52

*4.1. Числовые типы*

При работе с числами таких типов нужно помнить, что сравнение двух чисел с пла-вающей 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.

* 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 исполь-зованию типа double precision. Если же при объявлении типа параметр не исполь-зуется, то это также будет равносильно использованию типа double precision.

Последним из числовых типов является тип serial. Однако он фактически реализо-ван 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 имя-таблицы ( имя-столбца serial );**

Эта команда эквивалентна следующей группе команд:

**CREATE SEQUENCE имя-таблицы\_имя-столбца\_seq; CREATE TABLE имя-таблицы (**

**имя-столбца integer NOT NULL**

**DEFAULT nextval( 'имя-таблицы\_имя-столбца\_seq' )**

**);**

**ALTER SEQUENCE имя-таблицы\_имя-столбца\_seq OWNED BY имя-таблицы.имя-столбца;**

53

*Глава 4. Типы данных СУБД PostgreSQL*

Для пояснения вышеприведенных команд нам придется немного забежать впе-ред. Одним из видов объектов в базе данных являются так называемые последова-тельности. 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 последо-вательности, 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, ко-гда в таблице поле типа serial является суррогатным первичным ключом, тогда нет необходимости указывать явное значение для вставки в это поле.

* 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 нуж-но учитывать предполагаемое число строк в таблице и частоту удаления и вставки строк, поскольку даже для небольшой таблицы может потребоваться большой диа-пазон, если операции удаления и вставки строк выполняются часто.

**4.2. Символьные (строковые) типы**

Стандартные представители строковых типов — это типы character varying(n)

* character(n), где параметр указывает максимальное число символов в строке, которую можно сохранить в столбце такого типа. При работе с многобайтовыми ко-дировками символов, например UTF-8, нужно учитывать, что речь идет о символах, а не о байтах. Если сохраняемая строка символов будет короче, чем указано в опреде-лении 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, уста-новленных при компиляции исходных текстов СУБД.

54

*4.2. Символьные (строковые) типы*

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)

* 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)

* том случае, когда таких символов в константе много, все выражение становится трудно воспринимать. На помощь может прийти использование удвоенного символа «$». Эти символы выполняют ту же роль, что и одинарные кавычки, когда в них за-ключается строковая константа. При использовании символов «$» в качестве ограни-чителей 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 символь-ной 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».

55

*Глава 4. Типы данных СУБД PostgreSQL*

PostgreSQL предлагает еще одно расширение стандарта SQL — строковые константы

* стиле языка C. Чтобы иметь возможность их использовать, нужно перед начальной одинарной кавычкой написать символ «E».

Например, для включения в константу символа новой строки нужно сделать так:

**SELECT E'PGDAY\n17';**

?column?

----------

PGDAY +

17

(1 строка)

При использовании C-стиля необходимо удваивать обратную косую черту, если тре-буется поместить ее в константу буквально. А для включения в содержимое констан-ты символа обратной кавычки можно либо удвоить ее, либо сделать так:

**SELECT E'PGDAY\'17';**

?column?

----------

PGDAY'17

(1 строка)

Подробно об использовании C-стиля написано в разделе документации 4.1.2.2 «Стро-ковые константы со спецпоследовательностями в стиле C».

**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 време-ни, 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 вво-да дат таков: «yyyy-mm-dd», где символы «y», «m» и «d» обозначают цифру года, ме-сяца и дня соответственно. PostgreSQL позволяет использовать и другие форматы

56

*4.3. Типы «дата/время»*

для ввода, например: «Sep 12, 2016», что означает 12 сентября 2016 года. При выво-де 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)

* в следующем примере используется другой формат ввода, но формат вывода оста-ется 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 сим-вольной строкой, мы использовали операцию *приведения типа*. В PostgreSQL она оформляется с использованием двойного символа «двоеточие» и имени того типа,

* которому мы приводим данное значение. Важно учесть, что при выполнении при-ведения типа производится проверка значения на соответствие формату целевого типа и множеству его допустимых значений.
* PostgreSQL предусмотрен целый ряд функций для работы с датами и временем. На-пример, 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, 'dd-mm-yyyy' );**

57

*Глава 4. Типы данных СУБД PostgreSQL*

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 ча-совой 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 фор-маты, например:

**SELECT '21:15'::time;**

При выводе СУБД дополнит введенное значение, в котором присутствуют только ча-сы 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;**

58

*4.3. Типы «дата/время»*

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 конфигураци-онном файле сервера PostgreSQL. В приведенном примере значение часового пояса равно +03, но если ваш компьютер находится в другом часовом поясе, то это значе-ние будет другим, например, для регионов Сибири оно может быть +08.

59

*Глава 4. Типы данных СУБД PostgreSQL*

* результате объединения типов даты и времени получается интегральный тип — временная отметка. Этот тип существует в двух вариантах: с учетом часового поя-са — 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 при-мер 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)

* рассмотренных примерах мы использовали синтаксис тип 'строка' для указа-ния конкретного типа простой литеральной константы. Имя типа 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 по-хоже 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 временной´ отметки (т. е. даты и времени в одном значении) служит функция current\_timestamp. Она также вызывается без указа-ния круглых скобок. Приведем пример ее использования.

**SELECT current\_timestamp;**

now

-------------------------------

2016-09-27 18:27:37.767739+03

(1 строка)

60

*4.3. Типы «дата/время»*

Здесь в выводе присутствует и часовой пояс: +03.

Оба типа — 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. На-пример, в расписании авиарейсов указывается местное время как для аэропорта от-правления, так и для аэропорта прибытия. Поэтому в таком случае нужно использо-вать 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 от-резка 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 ка-честве 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. При-ведем 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).

61

*Глава 4. Типы данных СУБД PostgreSQL*

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 временными´ отметками, то значение этого типа можно получить при вычитании одной временной´ отметки из другой.

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

interval

----------

1. days

(1 строка)

Как мы уже говорили ранее, в PostgreSQL предусмотрен целый ряд функций для ра-боты с датами и временем. Например, для получения значений текущей даты, теку-щего времени и текущей временной´ 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 таб-лицы. Их особенностью является то, что при их вызове круглые скобки не исполь-зуются. Для получения полной информации обратитесь к документации (раздел 9.9 «Операторы и функции даты/времени»).

Значения временных´ отметок можно усекать с той или иной точностью с помощью функции date\_trunc. Например, для получения текущей временной´ отметки с точ-ностью 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 временных´ отметок можно с помощью функции extract извлекать от-дельные 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' );**

62

*4.4. Логический тип*

date\_part

-----------

11

(1 строка)

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

**4.4. Логический тип**

Логический (boolean) тип может принимать три состояния: истина и ложь, а так-же неопределенное состояние, которое можно представить значением NULL. Таким образом, тип boolean реализует трехзначную логику.

* качестве истинного состояния могут служить следующие значения: TRUE, 't',

'true', 'y', 'yes', 'on', '1'.

* качестве ложного состояния могут служить следующие значения: FALSE, 'f',

'false', 'n', 'no', 'off', '0'.

Для примера создадим таблицу с двумя столбцами и добавим в нее несколько строк следующим образом:

**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)

63

*Глава 4. Типы данных СУБД PostgreSQL*

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 содер-жаться не скалярные значения, а массивы переменной длины. Эти массивы могут быть многомерными и могут содержать значения любого из встроенных типов, а так-же типов данных, определенных пользователем.

Предположим, что нам необходимо сформировать и сохранить в базе данных в удоб-ной 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 наимено-ванию типа данных. При этом задавать число элементов не обязательно.

Давайте добавим в таблицу четыре строки. Массив в команде вставки представлен

* виде строкового литерала с указанием типа данных и квадратных скобок, означаю-щих массив. Обратите внимание, что все массивы имеют различное число элементов.

64

*4.5. Массивы*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **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 функ-ции 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

65

*Глава 4. Типы данных СУБД PostgreSQL*

* 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

* 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 необходи-мости 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

* 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). Та-ким 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)

66

*4.5. Массивы*

Теперь продемонстрируем основные операции, которые можно применять к масси-вам, выполняя выборки из таблиц. Получим список пилотов, летающих по средам:

**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 ука-занным значением в массив. Если же такого элемента нет, она возвратит NULL. Вы-берем 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 эле-менты правого массива. Конечно, при этом в левом массиве могут находиться и дру-гие элементы, что мы и видим в графике этого пилота.

Еще аналогичный вопрос: кто летает по вторникам и/или по пятницам? Для полу-чения ответа воспользуемся оператором &&, который проверяет наличие общих эле-ментов 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 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)

67

*Глава 4. Типы данных СУБД PostgreSQL*

Сформулируем вопрос в форме отрицания: кто не летает ни во вторник, ни в пятни-цу? 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 зна-чений, которые представлены в формате JSON (JavaScript Object Notation). Существу-ет два типа: json и jsonb. Основное различие между ними заключается в быстродей-ствии. Если столбец имеет тип json, тогда сохранение значений происходит быстрее, потому что они записываются в том виде, в котором были введены. Но при последу-ющем 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 дан-ного типа. Но все последующие обращения к сохраненным значениям выполняются быстрее, т. к. выполнять их разбор уже не требуется.

68

*4.6. Типы JSON*

Есть еще ряд отличий, в частности, тип json сохраняет порядок следования ключей

* объектах и повторяющиеся значения ключей, а тип jsonb этого не делает. Реко-мендуется 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 специ-альную 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

69

*Глава 4. Типы данных СУБД PostgreSQL*

**SELECT \* FROM pilot\_hobbies;**

|  |  |  |
| --- | --- | --- |
| pilot\_name | | | 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 авиа-компании для участия в турнире. Мы можем выбрать всех футболистов таким спосо-бом:

**SELECT \* FROM pilot\_hobbies**

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

pilot\_name | 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)

70

*4.6. Типы JSON*

* этом решении мы выводим только информацию о спортивных предпочтениях пи-лотов. 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 струк-туру объектов, т. е. конкретные имена ключей. Поэтому в принципе возможна ситуа-ция, когда в разных строках в JSON-объектах будут использоваться различные набо-ры ключей. В нашем примере структуры JSON-объектов во всех строках совпадают.

* если бы они не совпадали, то как можно было бы проверить наличие ключа? Про-демонстрируем 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)

* 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

71

*Глава 4. Типы данных СУБД PostgreSQL*

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 функ-ции 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 до-бавить новое значение. В данном случае этот путь состоит из имени ключа (sports)

* номера добавляемого элемента в массиве видов спорта (номер 1). Нумерация эле-ментов 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 | 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».

72

*Контрольные вопросы и задания*

**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 вво-димого 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, количе-ство цифр в котором слева от десятичной точки (запятой) превышает допус-тимое.

1. 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**

**);**

73

*Глава 4. Типы данных СУБД PostgreSQL*

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

**DROP TABLE test\_numeric;**

Вставьте в таблицу несколько строк:

**INSERT INTO test\_numeric**

**VALUES ( 1234567890.0987654321,**

**'Точность 20 знаков, масштаб 10 знаков' );**

**INSERT INTO test\_numeric**

**VALUES ( 1.5,**

**'Точность 2 знака, масштаб 1 знак' );**

**INSERT INTO test\_numeric**

**VALUES ( 0.12345678901234567890,**

**'Точность 21 знак, масштаб 20 знаков' );**

**INSERT INTO test\_numeric**

**VALUES ( 1234567890,**

**'Точность 10 знаков, масштаб 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.

1. Тип данных numeric поддерживает специальное значение NaN, которое озна-чает «не число» (not a number). В документации утверждается, что значение NaN считается равным другому значению NaN, а также что значение NaN считается большим´ 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)

1. 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.

74

*Контрольные вопросы и задания*

Например, сравним два очень маленьких числа (они представлены в экспонен-циальной 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 чис-лами, находящимися на границе допустимого диапазона для чисел типов real

и double precision.

1. 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. Напри-мер, 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.

75

*Глава 4. Типы данных СУБД PostgreSQL*

1. 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 од-ного из ее вариантов служит результат операции умножения нуля на бесконеч-ность. Посмотрите, что выдаст в результате PostgreSQL:

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

?column?

----------

NaN

(1 строка)

* документации утверждается, что значение NaN считается равным другому значению NaN, а также что значение NaN считается большим´ 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)

1. Тип serial может применяться для столбцов, содержащих числовые значения, которые должны быть уникальными в пределах таблицы, например, идентифи-каторы каких-то объектов. В качестве иллюстрации 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 ука-зывать 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.

76

*Контрольные вопросы и задания*

**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' );**

* теперь добавьте еще одну строку, но уже не указывая явно значение для столб-ца 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 автомати-ческое генерирование значений этого столбца.

1. Немного усложним определение таблицы из предыдущего задания. Пусть те-перь 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 уда-ления одной строки из нее. Для пошагового управления этим процессом выпол-няйте 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' );**

77

*Глава 4. Типы данных СУБД PostgreSQL*

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 фор-мировании нового значения из последовательности поиск максимального зна-чения, уже имеющегося в столбце, не выполняется.

id | name

----+------------

* | Вишневая

2 | Прохладная

3 | Грушевая

5 | Луговая

(4 строки)

1. Какой календарь используется в PostgreSQL для работы с датами: юлианский или григорианский?
2. Каждый тип данных из группы «дата/время» имеет ограничение на минималь-ное и максимальное допустимое значение. Найдите в документации в разделе 8.5 «Типы даты/времени» эти значения и подумайте, почему они таковы.
3. 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 за-проса: в первом запросе вообще не используем параметр точности, во втором назначим его равным 0, в третьем запросе сделаем его равным 3.

78

*Контрольные вопросы и задания*

**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 конфигурационного параметра datestyle. Значение этого параметра состоит из двух компонен-тов: первый управляет форматом вывода даты, а второй регулирует порядок следования составных частей даты (год, месяц, день) при вводе и выводе. Те-кущее 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 данных date

* timestamp. Для экспериментов возьмем дату, в которой число (день) превы-шает 12, чтобы нельзя было день перепутать с номером месяца. Пусть это будет, например, 18 мая 2016 г.

79

*Глава 4. Типы данных СУБД PostgreSQL*

**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;**

* 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.
* 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, незави-симо 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.

80

*Контрольные вопросы и задания*

Сейчас выберем третий способ, а первые два рассмотрим при выполнении дру-гих заданий. Поскольку параметр datestyle состоит фактически из двух ча-стей, которые можно задавать не только обе сразу, но и по отдельности, из-меним 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 замени-те 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 зна-чения 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 толь-ко при вводе значений, но и при выводе. Напомним, что вводом мы считаем команду SELECT, а выводом — результат ее выполнения, выведенный на экран.

* документации (см. раздел 8.5.2 «Вывод даты/времени») сказано, что формат вывода даты может принимать значения ISO, Postgres, SQL и German. Пер-вые 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 функ-циями current\_date и current\_timestamp.

81

*Глава 4. Типы данных СУБД PostgreSQL*

1. 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 нуж-но сделать это только на время текущего сеанса работы клиентской программы, например утилиты psql, то можно ввести значение этой переменной непосред-ственно в командной строке:

**PGDATESTYLE="Postgres" psql -d test -U имя-пользователя**

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

1. Назначить значение параметра datestyle можно в конфигурационном файле postgresql.conf, который находится в каталоге /usr/local/pgsql/data. Предвари-тельно сохраните текущую (корректно работающую) версию этого файла, а за-тем 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;**

1. В документации в разделе 9.8 «Функции форматирования данных» представ-лены 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,

* 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)

82

*Контрольные вопросы и задания*

**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.

1. 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.

1. 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;

^

1. 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.

83

*Глава 4. Типы данных СУБД PostgreSQL*

1. С типами даты и времени можно выполнять различные арифметические опера-ции. 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 предположе-ния с помощью утилиты psql. Подробное описание всех допустимых арифмети-ческих операций с датами и временем приведено в документации в разделе 9.9 «Операторы и функции даты/времени».

1. Значение типа interval можно получить при вычитании одной временной´ от-метки из другой, например:

**SELECT ( current\_timestamp - '2016-01-01'::timestamp )**

**AS new\_date;**

new\_date

-------------------------

278 days 00:10:33.33236

(1 строка)

* что получится, если прибавить интервал к временной´ отметке? Сначала по-пробуйте дать ответ, не прибегая к помощи утилиты psql, а затем проверьте свой ответ с помощью этой утилиты. Например, прибавим интервал длитель-ностью в 1 месяц к текущей к временной´ 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.

1. Можно с высокой степенью уверенности предположить, что при прибавлении интервалов к датам и временным´ 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 обос-нованные предположения о результатах следующих двух команд, а затем про-верьте 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;**

84

*Контрольные вопросы и задания*

1. 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 раз-личными 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 по-мощью команды

**SET intervalstyle TO DEFAULT;**

1. Выполните следующие две команды и объясните различия в выведенных ре-зультатах:

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

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

1. Выполните следующие две команды и объясните различия в выведенных ре-зультатах:

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

Почему при выполнении первой команды возникает ошибка? Как можно моди-фицировать эту команду, чтобы ошибка исчезла?

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

1. Значения временных´ отметок можно усекать с той или иной точностью с помо-щью 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)

85

*Глава 4. Типы данных СУБД PostgreSQL*

Напомним, что в данной команде используется операция приведения типа.

Выполните эту команду, последовательно указывая в качестве первого пара-метра значения microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium (которые обозначают соответственно микро-секунды, 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 округ-ления 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 использова-нии параметра week картина получается более интересная.

1. Функция date\_trunc может работать не только с данными типа timestamp, но также и с данными типа interval. Самостоятельно ознакомьтесь с этими возможностями по документации (см. раздел 9.9 «Операторы и функции да-ты/времени»).
2. Весьма полезной является функция extract. С ее помощью можно извлечь зна-чение 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 дополни-тельно преобразует число секунд в микросекунды и добавляет значение поля микросекунд.

date\_part

-----------

56123459

(1 строка)

Выполните эту команду, последовательно указывая в качестве первого пара-метра значения microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium. Можно использовать сокращения этих наиме-нований, которые приведены в предыдущем задании.

86

*Контрольные вопросы и задания*

Обратите внимание, что в ряде случаев выводится не просто конкретное по-ле (фрагмент) из временной´ отметки, а некоторый продукт переработки этого поля. Например, если в качестве первого параметра функции extract в выше-приведенной команде указать cent (век), то мы получим в ответ не 19 (что и было бы буквальным значением поля «век»), а 20, поскольку 1999 год принад-лежит двадцатому веку.

1. Функция extract может работать не только с данными типа timestamp, но также и с данными типа interval. Самостоятельно ознакомьтесь с этими возможностями по документации (см. раздел 9.9 «Операторы и функции да-ты/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?

87

*Глава 4. Типы данных СУБД PostgreSQL*

**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;**

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

|  |  |  |
| --- | --- | --- |
| person | | | birthday |
| -------------- | + | ------------ |
| Ken Thompson | | 1955-03-23 | |
| Ben Johnson | | 1971-03-19 | |

(2 строки)

88

*Контрольные вопросы и задания*

Если нам потребуется выяснить, кто из этих людей достиг возраста, скажем, 40 лет на момент выполнения запроса, то команда может быть такой (в послед-нем 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;**

* вот если мы захотим определить точный возраст каждого человека на теку-щий 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 показы-вает возраст в днях, а для пересчета числа дней в число лет нужны дополнитель-ные действия. Хотя, наверное, возможны ситуации, когда требуется определить возраст именно в днях.

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

89

*Глава 4. Типы данных СУБД PostgreSQL*

1. Изучая приемы работы с массивами, можно, как и в других случаях, пользовать-ся 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 об-разом:

**SELECT array\_remove( ARRAY[ 1, 2, 3 ], 3 );**

array\_remove

--------------

{1,2}

(1 строка)

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

33.\* В разделе документации 8.15 «Массивы» сказано, что массивы могут быть мно-гомерными 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[]**

**);**

90

*Контрольные вопросы и задания*

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)

91

*Глава 4. Типы данных СУБД PostgreSQL*

Предположим, что руководство авиакомпании решило, что пища пилотов долж-на 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,

* именно: столбец meal теперь будет содержать двумерные массивы. Опреде-ление этого столбца станет таким: meal text[][].

**Задание.** Создайте новую версию таблицы и соответственно измените коман-ду 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 эле-ментам двумерного массива нужно использовать два индекса. Не забывайте, что по умолчанию номера индексов начинаются с единицы.

1. В тексте раздела 4.6 мы выполняли обновление JSON-объекта с помощью функ-ции jsonb\_set: добавляли значение в массив. Для обновления скалярных зна-чений, 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 пред-ставляет 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)

92

*Контрольные вопросы и задания*

**Задание.** Самостоятельно выполните изменение значения по ключуhome\_lib

в одной из строк таблицы.

1. Изучая приемы работы с типами JSON, можно, как и в случае с массивами, поль-зоваться способностью команды SELECT обходиться без создания таблиц.

Покажем лишь один пример. Добавить новый ключ и соответствующее ему зна-чения в уже существующий объект можно 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 опе-раторов, 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 клю-чей. Добавьте дополнительный ключ и соответствующее ему значение в JSON-объект какой-нибудь строки таблицы pilots. Используйте оператор ||.

1. 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 опе-ратором -.

93

**Глава 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.

* 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**

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

Сначала представим описание таблицы «Студенты» (students). Она имеет следую-щую 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. зада-ние 10 в конце главы).

95

*Глава 5. Основы языка определения данных*

Теперь перейдем к таблице «Успеваемость» (progress). Ее структура 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 |
|  |  |  |  | term = 1 OR term = 2 |
| Оценка | mark | Числовой | numeric(1) | DEFAULT 5 |
|  |  |  |  | mark >= 3 AND |
|  |  |  |  | mark <= 5 |

* описаниях таблиц «Студенты» и «Успеваемость» есть колонки «Ограничения». Они уже содержат конкретные сведения, хотя ограничения мы еще детально не рассмат-ривали. Таким образом, мы немного забежали вперед, но по мере рассмотрения огра-ничений вам будет становиться понятно назначение каждого из них в обеих таб-лицах.

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 «Успевае-мость» (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 сло-во CHECK, за которым следует выражение в круглых скобках, содержащее одно или несколько условий, налагаемых на атрибуты таблицы. Это ограничение бывает двух видов: уровня атрибута и уровня таблицы. Различие между ними только в синтакси-ческом оформлении: в обоих случаях в выражении могут содержаться обращения не только к одному, но и к нескольким атрибутам таблицы. В первом случае ограниче-ние CHECK является частью определения одного конкретного атрибута, а во втором случае оно записывается как самостоятельный элемент определения таблицы.

96

*5.1. Значения по умолчанию и ограничения целостности*

Каждое ограничение имеет имя. Мы можем задать его сами с помощью ключевого слова CONSTRAINT. Если же мы этого не сделаем, тогда СУБД сформирует имя автома-тически. 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.

* качестве примера приведем ограничения, налагаемые на атрибуты term и mark из таблицы «Успеваемость». 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 ), ...**

**);**

* 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, ко-торые облегчат понимание причин появления сообщений об ошибках.
* 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 **ограничение** **NOT** **NULL.** Оно означает, что в столбце таблицы, на который наложено это ограничение,должны обязательно присутствовать какие-либо определенные значения. При разра-ботке баз данных, исходя из логики конкретной предметной области, зачастую требу-ется использовать это ограничение. Как сказано в документации, оно функциональ-но эквивалентно ограничению CHECK ( column\_name IS NOT NULL) , но в PostgreSQL создание явного ограничения NOT NULL является более эффективным подходом.

97

*Глава 5. Основы языка определения данных*

Еще один вид ограничений — это **ограничение уникальности** **UNIQUE.** Такое огра-ничение, наложенное на конкретный столбец, означает, что все значения, содержа-щиеся в этом столбце в различных строках таблицы, должны быть уникальными,

* е. не должны повторяться. Ограничение уникальности может включать в себя и несколько столбцов. В этом случае уникальной должна быть уже комбинация их зна-чений.

Когда в ограничение уникальности включается только один столбец, то можно задать ограничение непосредственно в определении столбца. Например, для таблицы «Сту-денты» 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 ),**

**...**

**CONSTRAINT unique\_record\_book UNIQUE ( record\_book ),**

**...**

**);**

Опять обратимся к таблице «Студенты» и покажем, как можно создать ограничение уникальности, включающее более одного столбца. В этой таблице первичным клю-чом является столбец record\_book, но очевидно, что комбинация значений серии и номера документа, удостоверяющего личность, является уникальной. Поэтому мож-но 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 ос-нове B-дерева для поддержки этого ограничения.

98

*5.1. Значения по умолчанию и ограничения целостности*

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 ре-альной работе заменять первичный ключ комбинацией ограничений UNIQUE и NOT NULL, поскольку теория баз данных требует наличия в каждой таблице именно пер-вичного ключа.

Первичный ключ является частью метаданных, его наличие позволяет другим табли-цам использовать его в качестве уникального идентификатора строк в данной таб-лице. 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 непосред-ственно 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 )**

**);**

* случае создания составного первичного ключа имена столбцов, входящих в его со-став, перечисляются в выражении PRIMARY KEY через запятую:

**PRIMARY KEY ( имя-столбца1, имя-столбца2, ...)**

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

* таблице может быть любое число ограничений UNIQUE, дополненных ограниче-нием NOT NULL, но первичный ключ может быть только один. PostgreSQL допускает и отсутствие первичного ключа, хотя строгая теория реляционных баз данных не ре-комендует так поступать.

99

*Глава 5. Основы языка определения данных*

Завершаем наш обзор различных видов ограничений рассмотрением такого важно-го 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 «Студен-ты» (students) и «Успеваемость» (progress). В первой из них содержатся данные

* студентах, а во второй — сведения об их успеваемости. Поскольку в процессе обу-чения 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 сту-дента 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

* таблице «Студенты» связаны. Для решения этой задачи не требуется в каждой стро-ке таблицы «Успеваемость» повторять все сведения о студенте: номер зачетной книж-ки, фамилию, имя и отчество, данные документа, удостоверяющего личность. Доста-точно включить в состав каждой строки таблицы «Успеваемость» лишь уникальный идентификатор строки из таблицы «Студенты». В нашем случае это будет номер за-четной 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», сопо-ставив значения атрибутов record\_book в строках обеих таблиц. В результате мы сможем получить все строки таблицы «Успеваемость», связанные с конкретной стро-кой из таблицы «Студенты» по внешнему ключу.

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

* 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 ),**

**...**

**);**

100

*5.1. Значения по умолчанию и ограничения целостности*

Предложение REFERENCES создает ограничение ссылочной целостности и указыва-ет в качестве ссылочного ключа атрибут record\_book. Это означает, что в таблицу «Успеваемость» нельзя ввести строку, значение атрибута record\_book которой от-сутствует в таблице «Студенты». 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 ат-рибутов:

**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 табли-цы и в первичном ключе ссылочной таблицы должны быть согласованы.

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

При наличии связей между таблицами, организованных с помощью внешних клю-чей, необходимо придерживаться определенной политики при выполнении опера-ций удаления и обновления строк в ссылочных таблицах — тех, на которые ссылаются другие таблицы. В нашем примере ситуация принятия «политического» решения воз-никает при удалении строк из таблицы «Студенты».

Конечно, если бы было принято решение хранить всю историю успеваемости сту-дентов, 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 «Успева-емость» (progress), которые ссылаются на удаляемую строку в таблице «Студенты» (students)?

101

*Глава 5. Основы языка определения данных*

Возможны несколько вариантов.

1. Удаление связанных строк из таблицы «Успеваемость», что означает, что при отчислении студента будет удаляться вся история его успехов в учебе. Эта опе-рация 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**

**);**

1. Запрет удаления строки из таблицы «Студенты», если в таблице «Успеваемость» есть хотя бы одна строка, ссылающаяся на удаляемую строку в таблице «Сту-денты». Для реализации такой политики в определение внешнего ключа до-бавляются 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. «Успевае-мость», 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 немед-ленно.

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**

**);**

102

*5.1. Значения по умолчанию и ограничения целостности*

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.

1. Присваивание атрибутам внешнего ключа в строках таблицы «Успеваемость» значения NULL. Для реализации этого подхода необходимо, чтобы на атрибу-ты 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**

**);**

1. Присваивание атрибутам внешнего ключа в строках таблицы «Успеваемость» значения по умолчанию (DEFAULT), если оно, конечно, было предписано при создании таблицы.

Оформляется этот вариант так (значение во фразе DEFAULT взято произволь-ным образом):

**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

103

*Глава 5. Основы языка определения данных*

ссылающейся таблицы, то будет иметь место нарушение ограничения ссылоч-ной целостности и операция удаления не будет выполнена.

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

**CREATE TABLE progress**

* **record\_book numeric( 5 ),**

**...**

**FOREIGN KEY ( record\_book )**

**REFERENCES students ( record\_book )**

**ON UPDATE CASCADE**

**);**

* случае каскадного обновления 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, которые можно указать для базы данных, мы можем привести окончательные определения таблиц «Студен-ты» и «Успеваемость». Окончательными они являются лишь в том смысле, что имен-но 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 )**

**);**

104

*5.2. Создание и удаление таблиц*

**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 «Авиапере-возки». 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 остав-шихся таблиц, но выполнять эти команды уже не требуется, достаточно будет только просматривать описания таблиц с помощью команды nd утилиты psql.

* главе 3 мы сначала представляли описание каждой создаваемой таблицы: назна-чения 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;**

105

*Глава 5. Основы языка определения данных*

Если вы уже были подключены к другой базе данных, то можете, не выходя из ути-литы psql, просто подключиться к нужной вам базе данных с помощью команды nconnect. Напомним, что команды, имена которых начинаются с символа «n», яв-ляются не SQL-командами, а командами утилиты psql. Конечно, за этими короткими командами, например nd, могут скрываться сложные SQL-запросы к системным таб-лицам 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, возмож-на ситуация, когда образуется иерархия таблиц. Таблица, находящаяся в середине такой иерархии, выполняет обе роли: ссылающейся и ссылочной таблицы. Тогда нуж-но 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 существу-ет таблица (или таблицы), которая не ссылается ни на какие другие таблицы. С нее и нужно начинать создание базы данных. Перед созданием очередной таблицы, име-ющей внешние 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 име-ет внешних ключей.

* этой таблице в качестве кода аэропорта служат трехбуквенные обозначения, утвер-жденные специальной организацией. При этом используются только буквы латин-ского алфавита. Каждый аэропорт имеет также и полное название. Оно не всегда совпадает с названием города, в котором аэропорт находится. Для города не преду-смотрено отдельной сущности, поэтому таблицы городов нет. Однако название го-рода 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

106

*5.2. Создание и удаление таблиц*

текстовый файл, содержащий команды для создания объектов базы данных, то ком-ментарии 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**

* команде nd 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**

* 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 мо-жете 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) |

107

*Глава 5. Основы языка определения данных*

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)

* этом выводе в выражении bookings.airports слово bookings означает имя **схе-мы**. Как мы уже говорили ранее, это, упрощенно говоря, раздел базы данных, в ко-тором 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 нашем случае — airports\_pkey. Оно было сгенери-ровано ядром PostgreSQL. Указан также и тип индекса — btree, т. е. B-дерево. Далее

* круглых скобках приводится список ключевых атрибутов. В нашем случае он состо-ит из одного атрибута — airport\_code.

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

PostgreSQL предлагает свое расширение — команду COMMENT, которая позволяет со-здавать комментарии (описания) к различным объектам базы данных. Эти коммен-тарии 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 nd добавить символ «+», например:

**\d+ airports**

Следующая таблица — **«Рейсы» (flights)**. Назначение ее атрибутов должно быть в це-лом понятно из комментариев, присутствующих в SQL-команде.

Сначала приведем саму команду, а затем сделаем ряд пояснений.

108

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | *5.2. Создание и удаление таблиц* |
| **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 ( status IN ( 'On Time', 'Delayed',** | | | **'Departed',** |
|  | **'Arrived', 'Scheduled', 'Cancelled' )** | | |
| **),** |  |  |  |
| **CHECK ( actual\_arrival IS NULL OR** | | |  |

* **actual\_departure IS NOT NULL AND actual\_arrival IS NOT NULL AND actual\_arrival > actual\_departure**

**)**

**),**

**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 )**

**);**

* таблице предусмотрено три внешних ключа, которые ссылаются на таблицы «Са-молеты» и «Аэропорты». В качестве первичного ключа используется так называемый **суррогатный ключ**, состоящий из одного атрибута —flight\_id. Обратите внима-ние, что тип данных этого атрибута — serial, т. е. значения целого типа для этого атрибута будут извлекаться из последовательности. Суррогатный ключ — это уни-кальный ключ, назначение которого — только идентифицировать строки в таблице. Зачастую для него используются целочисленные значения. Такому ключу не соот-ветствует никакое свойство никакой сущности реального мира. Это — абстракция, позволяющая в ряде случаев упростить определения таблиц, например, за счет со-кращения числа атрибутов во внешних ключах до одного. В нашей таблице «Рейсы» суррогатный ключ как раз и служит для того, чтобы в таблицах, ссылающихся на нее,

109

*Глава 5. Основы языка определения данных*

внешние ключи состояли только из атрибута flight\_id.

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 значе-ний: очевидно, что в один и тот же момент времени не могут выполняться два (и бо-лее) 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 важ-но, т. к. перелеты могут совершаться между городами, находящимися в разных часо-вых поясах, а время вылета и время прилета указываются местные.

Поясним смысл каждого из трех ограничений CHECK. Первое ограничение говорит

* 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 CHECK задает множество допустимых значений атрибута status следующим списком:

– Scheduled — рейс доступен для бронирования (это происходит за месяц до плано-вой 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.

Третье ограничение более сложное. Его можно условно разделить на две части, соеди-ненные логической операцией «ИЛИ». Первая часть говорит о том, что если самолет *еще не прилетел* (т. е. значениеactual\_arrivalне определено), то фактическое вре-мя 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 не определено, невозможно. Речь идет о сравнении вида > или <.

110

*5.2. Создание и удаление таблиц*

Вторая часть этого ограничения должна гарантировать, что если самолет *уже при-летел*, то, во-первых, фактическое время вылета должно быть отлично от NULL,

* во-вторых, фактическое время прилета должно быть больше фактического времени вылета.

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.

* частности, обратите внимание, что для атрибута flight\_id указан тип данных integer, а не serial, как предписано в команде для создания этой таблицы. В гла-ве 4 при рассмотрении типа данных serial мы говорили, ссылаясь на документа-цию, 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 последовательно-сти, говорит фраза

DEFAULT nextval('flights\_flight\_id\_seq'::regclass)

* этой фразе указано и имя последовательности — flights\_flight\_id\_seq. Если выполнить команду nd, 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 nd:

**\d flights\_flight\_id\_seq**

111

*Глава 5. Основы языка определения данных*

* базе данных есть еще одна таблица, не имеющая внешних ключей, — **«Бронирова-ния» (bookings)**. Это довольно простая таблица. В ней всего три атрибута.

Атрибут «Номер бронирования» (book\_ref) является первичным ключом. Посколь-ку 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.

* главе 4 мы уже говорили о том, что в случаях, требующих точных вычислений, необ-ходимо использовать числа с фиксированной точностью. Работа с денежными сум-мами как раз и является одним из таких случаев. Поэтому для атрибута «Полная сто-имость 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 )**

**);**

* таблицей «Бронирования» (bookings) по внешнему ключу связана таблица **«Биле-ты» (tickets)**.
* качестве первичного ключа в ней 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).
* качестве идентификатора пассажира будет использоваться номер документа, удо-стоверяющего 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 со-здавать дополнительные атрибуты в таблице нецелесообразно. С такими данными —

112

*5.2. Создание и удаление таблиц*

их называют слабоструктурированными — PostgreSQL хорошо умеет работать: для них предусмотрены типы json и jsonb. В нашей таблице используется тип jsonb

* вот почему: хотя ввод данных такого типа несколько замедляется из-за необходи-мости 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 в таблице будет атрибут «Номер бронирования» (book\_ref), по-скольку в рамках каждой процедуры бронирования может быть оформлено более од-ного 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 о всех перелетах хранится 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

* 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).
* каждым перелетом связан класс обслуживания, значения этого атрибута подлежат проверке с помощью ограничения CHECK. Точно такое же ограничение есть и в таб-лице «Места» (seats), в которой каждому месту в салоне конкретного типа самолета присваивается определенный класс обслуживания.

Атрибут «Стоимость перелета» (amount) требует использования типа данных numer-ic, поскольку, как мы уже говорили ранее, денежные суммы должны записываться

* 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 внеш-ними ключами.

113

*Глава 5. Основы языка определения данных*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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 би-летов на конкретный рейс, поэтому тип данных выбирается integer.

Обратите внимание, что эта таблица имеет связь типа 1:1 с таблицей «Перелеты». Это объясняется тем, что пассажир, купивший билет на конкретный рейс, при регистра-ции получает только один посадочный талон. Конечно, если пассажир не явился на регистрацию, он не получает талона. Поэтому число строк в таблице «Посадочные талоны» может в общем случае оказаться меньше числа строк в таблице «Переле-ты». Логично ожидать, что первичные ключи у этих двух таблиц будут одинаковы-ми: 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 регистра-ции 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 пас-сажиров, поэтому создается уникальный ключ с атрибутами «Идентификатор рейса» (flight\_id) и «Номер места» (seat\_no). Еще один уникальный ключ призван га-рантировать несовпадение номеров посадочных талонов на данном рейсе, он вклю-чает атрибуты «Идентификатор рейса» (flight\_id) и «Номер посадочного талона» (boarding\_no).

114

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | *5.2. Создание и удаление таблиц* |
| **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**

* процессе создания таблиц между ними образовывались связи за счет внешних клю-чей. 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 вос-создать таблицы базы данных будет совсем нетрудно, поэтому вы можете смело вы-полнять команды удаления таблиц.

Давайте сначала попытаемся удалить таблицу aircrafts:

**DROP TABLE aircrafts;**

115

*Глава 5. Основы языка определения данных*

Казалось бы, не должно быть никаких проблем, но в результате СУБД выдает сооб-щение 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 этом сообщении. Выполнив команду

**\d flights**

мы увидим внешний ключ, ссылающийся на таблицу «Самолеты». В сообщении со-держится также и подсказка, рекомендующая в команду DROP TABLE добавить фразу CASCADE, означающую каскадное удаление зависимых объектов. Давайте так и сде-лаем:

**DROP TABLE aircrafts CASCADE;**

Теперь удаление таблицы прошло успешно, при этом из таблиц «Рейсы» и «Места» 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

* 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;**

116

*5.3. Модификация таблиц*

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 со-общений 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 выпол-няется 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. зна-чение «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 си-туации, 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 стол-бец из таблицы — для этого есть фраза DROP COLUMN. Если нужно добавить ограни-чение, то помогут фразы ADD CHECK и ADD CONSTRAINT. Если потребовался внешний ключ, то можно добавить и его.

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

Предположим, что нам понадобилось иметь в базе данных сведения о крейсер-ской скорости полета всех моделей самолетов, которые эксплуатируются в нашей авиакомпании. Следовательно, необходимо добавить столбец в таблицу «Самолеты»

117

*Глава 5. Основы языка определения данных*

(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 ограничение и на минимальное значение крейсер-ской 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» уже есть строки. Однако во время добавления тех строк столбец speed

* 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 ново-му ограничению.

Как же можно выйти из этой ситуации? Один из вариантов такой: сначала добавить столбец, не накладывая на его значения никаких ограничений, затем ввести зна-чения нового атрибута в уже существующие строки, причем эти значения должны удовлетворять тем ограничениям, которые мы собираемся наложить. После этого на-кладываем 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**

118

*5.3. Модификация таблиц*

Конечно, если необходимость наличия того или иного ограничения отпадет, его мож-но 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 «Координа-ты аэропорта: долгота» (longitude) и «Координаты аэропорта: широта» (latitude)

* float (double precision) на numeric(5, 2). Сделать это можно с помощью од-ной команды, поскольку команда ALTER TABLE поддерживает и выполнение более одного действия за один раз.

Сначала посмотрим, с какой точностью выводятся значения этих атрибутов до изме-нения типа данных, затем изменим тип данных для двух столбцов, опять выведем содержимое таблицы на экран и убедимся, что значения были округлены в соответ-ствии 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;**

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

119

*Глава 5. Основы языка определения данных*

Однако если исходный и целевой типы данных относятся к разным группам, тогда потребуются некоторые дополнительные усилия с нашей стороны. В качестве при-мера рассмотрим следующую ситуацию. Предположим, что по результатам опытной эксплуатации базы данных «Авиаперевозки» мы пришли к выводу о том, что необ-ходимо 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», а именно: тип дан-ных 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

120

*5.3. Модификация таблиц*

* в самом деле, в определении таблицы «Места» 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 до-бавить 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» и «Классы обслужива-ния» по внешнему ключу. Сделаем это:

**ALTER TABLE seats**

**ADD FOREIGN KEY ( fare\_conditions )**

**REFERENCES fare\_conditions ( fare\_conditions\_code );**

Посмотрев описание таблицы «Места», 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 ссылать-ся только на одноименные атрибуты ссылочной таблицы. Сейчас мы на практике успешно проверили это утверждение. Однако для удобства сопровождения базы дан-ных имеет смысл переименовать столбец fare\_conditions в таблице «Места», т. е.

121

*Глава 5. Основы языка определения данных*

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 заметить, что имя атрибута, являющегося внешним ключом, изменилось, а вот имя ограничения осталось неиз-менным (seats\_fare\_conditions\_fkey), хотя оно и было первоначально сформи-ровано 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 име-нования ограничений:

**ALTER TABLE seats**

**RENAME CONSTRAINT seats\_fare\_conditions\_fkey**

**TO seats\_fare\_conditions\_code\_fkey;**

Как всегда, проверим, что получилось:

**\d seats**

* в заключение этого раздела вернемся к таблице «Классы обслуживания». Мы преду-смотрели в ней первичный ключ, но ведь значения атрибута «Наименование класса обслуживания» (fare\_conditions\_name) также должны быть уникальными, дубли-рование 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**

122

*5.4. Представления*

**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 несколь-ким таблицам. Чтобы избежать необходимости многократного формирования таких запросов, можно использовать так называемые представления (views). Если речь идет

* выборке данных, то представления практически неотличимы от таблиц с точки зре-ния обращения к ним в командах SELECT.

Упрощенный синтаксис команды CREATE VIEW, предназначенной для создания представлений, таков:

**CREATE VIEW имя-представления [ ( имя-столбца [, ...] ) ] AS запрос;**

* этой команде обязательными элементами являются имя представления и запрос к базе данных, который и формирует выборку из нее. Если список имен столбцов не указан, 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.

123

*Глава 5. Основы языка определения данных*

**SELECT \* FROM seats\_by\_fare\_cond;**

* отличие от таблиц, представления не содержат данных. При каждом обращении к представлению в команде SELECT данные выбираются из таблиц, на основе которых это представление создано.

СУБД PostgreSQL предлагает свое расширение команды CREATE VIEW, а именно — фразу OR REPLACE. Если представление уже существует, то можно его не удалять,

* 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 представле-ние 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 указании 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

* 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 спис-ка имен столбцов, приводимого в начале команды. Так что если мы все же захотим

124

*5.4. Представления*

изменить имя столбца в представлении, нам придется сначала удалить это представ-ление, 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;**

* 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, кото-рый лежит в основе конкретного представления, может измениться. При этом для прикладного программиста представление останется неизменным, поэтому не по-требуется переделывать запросы к этому представлению в прикладной программе.

* + базе данных «Авиаперевозки» создано представление «Рейсы» (flights\_v), скон-струированное на основе таблицы «Рейсы» (flights), но содержащее дополнитель-ную 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);

125

*Глава 5. Основы языка определения данных*

– 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 при-ведем 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, | scheduled\_departure\_local | timestamp |
| местное время в пункте отправления |  |  |
| Время прилета по расписанию | scheduled\_arrival | timestamptz |
| Время прилета по расписанию, | scheduled\_arrival\_local | timestamp |
| местное время в пункте прибытия |  |  |
| 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, | actual\_departure\_local | timestamp |
| местное время в пункте отправления |  |  |
| Фактическое время прилета | actual\_arrival | timestamptz |
| Фактическое время прилета, | actual\_arrival\_local | timestamp |
| местное время в пункте прибытия |  |  |
| Фактическая продолжительность полета | actual\_duration | interval |

Известно, что в сфере железнодорожных пассажирских перевозок время в расписа-нии движения поездов и в билетах указывается московское. А в пассажирских авиа-

126

*5.4. Представления*

перевозках, напротив, время в билетах указывается местное. Это касается и времени вылета и времени прилета. Если пункты отправления и назначения находятся в раз-личных 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 столб-ца, отображающие местное время: два из них относятся к пункту отправления — scheduled\_departure\_local и actual\_departure\_local, а два других относят-ся к пункту прибытия — scheduled\_arrival\_local и actual\_arrival\_local.

* 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, установлен-ному на компьютере пользователя, а нам нужно сохранить его значения такими, ка-кими они являются в пункте отправления и пункте назначения.

Для перевода значения типа timestamptz (с часовым поясом) в значение типа timestamp (без часового пояса) служит конструкция AT TIME ZONE, подробно рас-смотренная в разделе 9.9 «Операторы и функции даты/времени» документации. Так-же существует и эквивалентная функция timezone, которая и используется здесь для пересчета московского времени в местное.

Если вы испытываете затруднения с пониманием операций преобразования значе-ний 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**

* представлении «Рейсы» 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 представ-ления «Рейсы».

127

*Глава 5. Основы языка определения данных*

|  |  |
| --- | --- |
| **SELECT \* FROM flights\_v;** |  |
| -[ 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 несуще-ствующего представления. В таких случаях обычно стараются избежать ненужных сообщений об ошибке отсутствия представления. Для этого в команду DROP VIEW до-бавляют фразу IF EXISTS. Например:

**DROP VIEW IF EXISTS flights\_v;**

Как мы уже говорили ранее, представление является фактически сохраненным за-просом к базе данных. Этот запрос получает имя, которым можно воспользоваться в предложении FROM команды SELECT для получения результатов этого запроса.

PostgreSQL предлагает свое расширение — так называемое материализованное пред-ставление. Упрощенный синтаксис команды CREATE MATERIALIZED VIEW, предна-значенной для создания материализованных представлений, таков:

**CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] имя-мат-представления**

* **( имя-столбца [, ...] ) ] AS запрос**
* **WITH [ NO ] DATA ];**

128

*5.4. Представления*

* момент выполнения команды создания материализованного представления оно заполняется данными, но только если в команде не было фразы WITH NO DATA. Ес-ли же она была включена в команду, тогда в момент своего создания представле-ние остается пустым, 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.

* нашей учебной базе данных «Авиаперевозки» имеется материализованное пред-ставление — «Маршруты» (routes). Как вы могли заметить, таблица «Рейсы» содер-жит избыточность: для одного и того же номера рейса, отправляющегося в различные дни, повторяются коды аэропортов отправления и назначения, а также код самоле-та. 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 «Маршруты», 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 выпол-няются рейсы». Это массив целых чисел.

Если впоследствии вам потребуется обновить данные в материализованном пред-ставлении, то выполните команду

**REFRESH MATERIALIZED VIEW routes;**

129

*Глава 5. Основы языка определения данных*

Кончено, как и любой другой объект базы данных, материализованное представле-ние можно удалить.

**DROP MATERIALIZED VIEW routes;**

Подводя итог раздела, назовем положительные стороны использования представле-ний.

1. Упрощение разграничения полномочий пользователей на доступ к хранимым данным.

Разным типам пользователей могут требоваться различные данные, хранящие-ся в одних и тех же таблицах. Это касается как столбцов, так и строк таблиц. Со-здание различных представлений для разных пользователей избавляет от необ-ходимости создавать дополнительные таблицы, дублируя данные, и упрощает организацию системы управления доступом к данным.

1. Упрощение запросов к базе данных.

Запросы к базе данных могут включать несколько таблиц и быть весьма слож-ными и громоздкими, при этом такие запросы могут выполняться часто. Ис-пользование представлений позволяет скрыть эти сложности от прикладного программиста и сделать запросы более простыми и наглядными.

1. Снижение зависимости прикладных программ от изменений структуры таблиц базы данных.
   * процессе развития информационной системы структура таблиц базы данных может изменяться. Столбцы представления, т. е. их имена, типы данных и по-рядок следования, — это, образно говоря, интерфейс к запросу, который реа-лизуется данным представлением. Если этот интерфейс остается неизменным, то SQL-запросы, в которых используется данное представление, корректиро-вать не потребуется. Нужно будет лишь в ответ на изменение структуры базовых таблиц, на основе которых представление сконструировано, соответствующим образом перестроить запрос, выполняемый данным представлением.
2. Снижение времени выполнения сложных запросов за счет использования мате-риализованных представлений.
   * материализованных представлениях можно сохранять результаты выполне-ния запросов, которые формируются длительное время, но при этом допускают их формирование заранее, а не обязательно в момент возникновения потребно-сти в результатах этого запроса. Если, например, какой-нибудь сводный отчет формируется длительное время, а запросы к отчету будут неоднократными, то

130

*5.5. Схемы базы данных*

может оказаться целесообразным сформировать его заранее и сохранить в ма-териализованном представлении.

Тем не менее нужно учитывать, что применимость материализованных пред-ставлений весьма ограничена. Не следует заменять ими все сложные запросы. Одним из 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 раз-личные 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.

* каждой базе данных может содержаться более одной схемы. Их имена должны быть уникальными в пределах конкретной базы данных. Имена объектов базы дан-ных (таблиц, представлений, последовательностей и др.) должны быть уникальными в пределах конкретной схемы, но в разных схемах имена объектов могут повторять-ся. 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)

* 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;**

131

*Глава 5. Основы языка определения данных*

Теперь объясним подробнее, что эта команда делает.

Если в базе данных создано более одной схемы, то доступ к объектам, содержащимся

* конкретной схеме, можно организовать разными способами. Первый заключается
* 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 преду-смотрены в конфигурации сервера PostgreSQL, есть параметр search\_path. Его зна-чение 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 схе-мы с именами, совпадающими с именами пользователей. Тогда могут упроститься некоторые операции с базой данных. Однако в базе данных demo нет таких схем, по-этому первый элемент параметра search\_path фактически не участвует в работе,

* результате все обращения к объектам базы данных без указания имени схемы бу-дут адресоваться схеме public.

Чтобы изменить порядок просмотра схем при поиске объектов в базе данных, нуж-но воспользоваться командой SET. При этом первой в списке схем следует указать именно ту, которую СУБД должна просматривать первой. Эта схема и станет теку-щей. 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;**

132

*Контрольные вопросы и задания*

search\_path

-------------

bookings

(1 строка)

Да, действительно, теперь первой будет просматриваться схема bookings. А для об-ращения к объектам, например, таблицам, в схеме public (если бы они в ней были) нам пришлось бы указывать имя схемы public перед именами этих объектов. Ес-ли 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 сле-дующее: 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 сде-лать так:

**CREATE TABLE my\_schema.airports**

**...**

**Контрольные вопросы и задания**

1. При использовании значений по умолчанию с ключевым словом DEFAULT воз-можны и ситуации, когда типичным будет не конкретное значение данных, а способ его получения. Например, если мы захотим фиксировать в каждой строке таблицы «Студенты» имя пользователя базы данных, добавившего эту строку в таблицу, тогда необходимо в определение таблицы добавить еще один

133

*Глава 5. Основы языка определения данных*

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,

* при вставке каждой строки. При этом в команде INSERT не требуется указы-вать 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 табли-цы 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 уда-лите ее, а затем создайте заново:

**DROP TABLE students;**

**CREATE TABLE students ...**

1. Посмотрите, какие ограничения уже наложены на атрибуты таблицы «Успевае-мость» (progress). Воспользуйтесь командой \d утилиты psql. А теперь пред-ложите 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 допусти-мых значений атрибута «Оценка» (mark) будет зависеть от того, экзамен или за-чет предусмотрены по данной дисциплине. Если предусмотрен экзамен, тогда допускаются значения 3, 4, 5, если зачет — тогда 0 (не зачтено) или 1 (зачтено).

134

*Контрольные вопросы и задания*

Не забудьте, что значения NULL для атрибутов test\_form и mark не допуска-ются.

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 таб-лице progress. Для этого выполните команды INSERT, как удовлетворяющие ограничению, так и нарушающие его.

* 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 ограниче-ние 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 во-прос: не является ли ограничение NOT NULL избыточным? Ведь в ограничении CHECK явно указаны 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).

1. В определении таблицы «Успеваемость» (progress) для атрибута mark не толь-ко задано ограничение CHECK, но и установлено значение по умолчанию с по-мощью ключевого слова DEFAULT:

**...**

**mark numeric( 1 ) NOT NULL**

**CHECK ( mark >= 3 AND mark <= 5 ) DEFAULT 5,**

**...**

135

*Глава 5. Основы языка определения данных*

Как вы думаете, что будет, если в ограничении DEFAULT мы «случайно» допу-стим ошибку, написав DEFAULT 6? Если в команде INSERT не указать значение для атрибута mark, то на каком этапе эта ошибка будет выявлена: уже на этапе создания таблицы или только при вставке строки в нее?

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 );**

1. В стандарте SQL сказано, что при наличии ограничения уникальности, вклю-чающего 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 огра-ничение уникальности по двум столбцам: doc\_ser и doc\_num. А затем про-верьте вышеприведенное утверждение, добавив в таблицу не только строки, со-держащие 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 совпадающи-ми. Это можно проверить с помощью команды

**SELECT (null = null);**

Она даст такой результат (т. е. NULL):

?column?

----------

(1 строка)

1. Модифицируйте определения таблиц «Студенты» (students) и «Успеваемость» (progress). В таблице students в качестве первичного ключа назначьте ком-бинацию атрибутов doc\_ser и doc\_num, а в таблице progress соответствую-щим образом измените определение внешнего ключа.

136

*Контрольные вопросы и задания*

**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 подра-зумевается 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 по-требуется, то и определение таблицы «Студенты» (students), чтобы изучить все варианты реагирования СУБД на обновление строк в ссылочной таблице,

* данном случае — students. Последовательно изменяйте определение внеш-него ключа таблицы progress, испробовав варианты ON UPDATE CASCADE, ON UPDATE RESTRICT, ON UPDATE SET NULL и ON UPDATE SET DEFAULT. Для получе-ния 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 зна-чение по умолчанию для атрибута внешнего ключа в ссылающейся таблице,

137

*Глава 5. Основы языка определения данных*

* во-вторых, это значение по умолчанию все равно должно присутствовать в одной из строк ссылочной таблицы. Как вы считаете, с учетом сказанного, воз-можно ли использование ON UPDATE SET DEFAULT в нашем случае?

Попробуйте обосновать или, наоборот, опровергнуть целесообразность исполь-зования каждой из этих политик — CASCADE, RESTRICT, SET NULL и SET DEFAULT — при выполнении операции UPDATE в реальной информационной системе, предназначенной для учета успеваемости студентов.

1. В таблице «Успеваемость» (progress) есть атрибут «Учебная дисциплина» (subject). Это текстовый атрибут. Одинаковые наименования учебных дисци-плин записываются в таблицу progress многократно. Создайте еще одну таб-лицу — «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 ка-честве 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 боль-ше 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 но-вой таблицей subjects.

1. В таблице «Студенты» (students) есть текстовый атрибут name, на который на-ложено 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 );**

138

*Контрольные вопросы и задания*

Наверное, проектируя эту таблицу, мы хотели бы все же, чтобы пустые строки

* качестве значения атрибута name не проходили в базу данных? Какое реше-ние вы можете предложить? Видимо, нужно добавить ограничение CHECK для столбца name. Если вы еще не изучили команду ALTER TABLE, то удалите табли-цу students и создайте ее заново с учетом нового ограничения, а если вы уже познакомились с командой ALTER TABLE, то сделайте так:

**ALTER TABLE students ADD CHECK ( name <> '' );**

Добавив ограничение, попробуйте теперь вставить в таблицу students строку (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 предло-жить? В разделе 9.4 документации «Строковые функции и операторы» есть функция trim. Попробуйте воспользоваться ею. Если вы еще не изучили коман-ду ALTER TABLE, то удалите таблицу students и создайте ее заново с учетом нового ограничения, а если уже познакомились с ней, то сделайте так:

**ALTER TABLE students ADD CHECK (...);**

Есть ли подобные слабые места в таблице «Успеваемость» (progress)?

1. В таблице «Студенты» (students) атрибут «Серия документа, удостоверяюще-го личность» (doc\_ser) имеет числовой тип, однако в сериях таких документов могут встречаться лидирующие нули, которые в числовых столбцах не сохраня-ются. Например, при записи значения серии «0402» первый ноль не сохранится.

Модифицируйте таблицу students, заменив числовой тип данных на символь-ный, например, character. Как вы думаете, эта операция пройдет без затруд-нений или они все же возможны? Проверьте ваши предположения, выполнив модификацию.

139

*Глава 5. Основы языка определения данных*

11.\* В таблице «Рейсы» (flights) есть ограничение, которое регулирует соотноше-ния 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 суще-ствующее 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 зна-чения 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.

1. Команда ALTER TABLE позволяет переименовать таблицу. Например:

**ALTER TABLE table\_name RENAME TO new\_table\_name;**

140

*Контрольные вопросы и задания*

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.

1. И представление «Рейсы» (flights\_v), и материализованное представление «Маршруты» (routes) построены на основе таблиц «Рейсы» (flights) и «Аэро-порты» (airports). Логично предположить, что при каскадном удалении, на-пример, таблицы «Аэропорты», представление «Рейсы» будет также удалено, поскольку при удалении базовой таблицы этому представлению просто неот-куда будет брать данные.

* что вы можете предположить насчет материализованного представления «Маршруты»: будет ли оно также удалено или нет? Ведь оно уже *содержит* дан-ные, 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 смо-жете 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.

1. Представления (views) могут быть обновляемыми. Это значит, что можно с по-мощью команд INSERT, UPDATE и DELETE, применяемых к представлению, вне-сти 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.

1. Определение таблицы можно изменить с помощью команды ALTER TABLE. Ана-логичные команды существуют и для изменения представлений и материали-зованных представлений: ALTER VIEW и ALTER MATERIALIZED VIEW. Самосто-ятельно ознакомьтесь с их возможностями с помощью документации.

141

*Глава 5. Основы языка определения данных*

1. Как вы думаете, при изменении данных в таблицах, на основе которых скон-струировано материализованное представление, содержимое этого представ-ления тоже синхронно изменяется или нет?

Если содержимое материализованного представления изменяется синхронно с базовыми таблицами, то продемонстрируйте это. Если же оно остается неиз-менным, то покажите, как его синхронизировать с базовыми таблицами.

1. 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 техниче-ских характеристиках самолетов, эксплуатируемых в авиакомпании. Пусть это будут такие сведения, как число членов экипажа (пилоты), тип двигателей и их количество.

142

*Контрольные вопросы и задания*

Следовательно, необходимо добавить новый столбец в таблицу «Самолеты» (aircrafts). Дадим ему имя specifications, а в качестве типа данных выбе-рем 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)

143

*Глава 5. Основы языка определения данных*

Чтобы получить еще более детальные сведения, например, о типе двигателей, нужно учитывать, что созданный JSON-объект имеет сложную структуру: он со-держит 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 дополнитьстолбцами типа json/jsonb. Вспомните, что, например, в таблице «Билеты» (tickets) уже есть столбец такого типа — contact\_data. Выполните модифи-кации таблиц и измените в них одну-две строки для проверки правильности ваших решений.

144

**Глава 6**

**Запросы**

Эта глава будет самой насыщенной и интересной, поскольку умение писать SQL-запросы — это не только ремесло, но, пожалуй, и искусство тоже.

* 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 вы-борках 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.
* этой главе мы покажем более сложные способы их получения.
* целью приведения в систему тех знаний о формировании выборок, что были получены в предыду-щих 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» содер-жит чуть больше ста строк. Для ее просмотра можно включить расширенный режим вывода данных nx.

**SELECT \* FROM aircrafts;**

**SELECT \* FROM airports;**

Начнем с различных условий отбора строк в предложении WHERE. Эти условия мо-гут конструироваться с использованием следующих **операторов сравнения**: =, < >, >, > =, <, < =. В предыдущих главах мы уже использовали ряд таких операторов, поэтому сейчас рассмотрим некоторые другие способы осуществления отбора строк.

145

*Глава 6. Запросы*

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 соответ-ствует любой последовательности символов, т. е. вместо него могут быть подстав-лены 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

---------------+-----------------+-------

1. | Airbus A320-200 | 5700
2. | Airbus A321-200 | 5600
3. | 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 при-меняется оператор LIKE, создан индекс для ускорения доступа к данным, то при наличии символа «%» в начале шаблона этот индекс использоваться не будет. Из-за этого может ухудшиться производительность, т. е. запрос будет выполняться медлен-нее. Индексы подробно рассматриваются в главе 8, а вопросы производительности — в главе 10.

Конечно, существует и оператор NOT LIKE. Например, если мы захотим узнать, каки-ми самолетами, кроме машин компаний Airbus и Boeing, располагает наша авиаком-пания, 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)

146

*6.1. Дополнительные возможности команды SELECT*

Кроме символа «%» в шаблоне может использоваться и символ подчеркивания — «\_», который соответствует в точности одному любому символу. В качестве примера най-дем в таблице «Аэропорты» те из них, которые имеют названия длиной три символа (буквы). С этой целью зададим в качестве шаблона строку, состоящую из трех симво-лов «\_».

**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 вы-брать, например, самолеты компаний Airbus и Boeing, можно сделать так:

**SELECT \* FROM aircrafts WHERE model ~ '^(A|Boe)';**

aircraft\_code | model | range

---------------+-----------------+-------

1. | Boeing 777-300 | 11100
2. | Boeing 767-300 | 7900
3. | Airbus A320-200 | 5700
4. | Airbus A321-200 | 5600
5. | Airbus A319-100 | 6700
6. | Boeing 737-300 | 4200

(6 строк)

Оператор ~ ищет совпадение с шаблоном с учетом регистра символов. Символ «^»

* 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
* круглых скобках означает альтернативный выбор между значениями, разделяемы-ми символом «|». Поэтому в выборку попадут значения, начинающиеся либо на «A», либо на «Boe».

Для инвертирования смысла оператора ~ нужно перед ним добавить знак «!». В каче-стве примера отыщем модели самолетов, которые не завершаются числом 300.

**SELECT \* FROM aircrafts WHERE model !~ '300$';**

147

*Глава 6. Запросы*

* этом регулярном выражении символ «$» означает привязку поискового шаблона к концу строки. Если же требуется проверить наличие такого символа *в составе* стро-ки, 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 доку-ментации 9.7.3 «Регулярные выражения POSIX».

* качестве замены традиционных операторов сравнения могут использоваться **пре-дикаты сравнения**, которые ведут себя так же, как и операторы, но имеют другойсинтаксис.

Давайте ответим на вопрос: какие самолеты имеют дальность полета в диапазоне от 3 000 км до 6 000 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 3 000 включено в полученную выборку.

При выборке данных можно проводить вычисления и получать в результирующей таблице **вычисляемые столбцы**. Если мы захотим представить дальность полета 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;**

148

*6.1. Дополнительные возможности команды SELECT*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 поряд-ка строк в результирующей выборке. Для упорядочения строк служит **предложение** **ORDER BY**, которое мы уже использовали ранее. Однако мы не говорили, что можнозадать не только возрастающий, но также и убывающий порядок сортировки. Напри-мер, 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 |
| 319 | | Airbus | | A319-100 | | | 6700 |
| 320 | | Airbus | | A320-200 | | | 5700 |
| 321 | | Airbus | | A321-200 | | | 5600 |
| 733 | | Boeing | | 737-300 | | | 4200 |
| SU9 | | | Sukhoi | Superjet-100 | | | 3000 |
| CR2 | | | Bombardier CRJ-200 | | | | 2700 |
| CN1 | | | Cessna | 208 Caravan | | | 1200 |

(9 строк)

149

*Глава 6. Запросы*

Мы детально разобрались с таблицей «Самолеты» и теперь обратим наше внимание на таблицу «Аэропорты»). В ней есть столбец «Часовой пояс» (timezone). Давайте по-смотрим, 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. Ко-нечно, 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 орга-низацией IANA (Internet Assigned Numbers Authority), и отличаются от традиционных

150

*6.1. Дополнительные возможности команды SELECT*

географических и административных часовых поясов, число которых в России равно одиннадцати.

* таблице «Аэропорты» 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 значе-ний 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)

* как найти еще три аэропорта, которые находятся немного западнее первой трой-ки, 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)

* дополнение к вычисляемым столбцам, когда выводимые значения получают пу-тем вычислений, при выборке данных из таблиц можно использовать **условные вы-ражения**, позволяющие вывести то или иное значение в зависимости от условий.

151

*Глава 6. Запросы*

* таблице «Самолеты» 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, среднема-гистральных или ближнемагистральных судов.

Для этого подойдет конструкция

**CASE WHEN условие THEN выражение**

**[ WHEN ... ]**

* **ELSE выражение ]**

**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**

* 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.

152

*6.2. Соединения*

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;**

* 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 зна-чений атрибутов**.

* 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 спис-ке 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.

153

*Глава 6. Запросы*

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». 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 ат-рибута 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 наиме-нование модели. Поэтому с помощью условия WHERE можно будет отобрать только те результирующие строки, в которых значение атрибута model будет «Cessna 208 Caravan».

* какие столбцы оставлять в списке столбцов предложения SELECT, решать нам. Даже если мы соединяем две таблицы (или более), то совершенно не обязательно в резуль-тирующий список столбцов включать столбцы всех таблиц, перечисленных в предло-жении 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 |
| 2A | | | Economy |
| 2B | | | Economy |
| 3A | | | Economy |
| 3B | | | Economy |
| 4A | | | Economy |
| 4B | | | Economy |
| 5A | | | Economy |

154

*6.2. Соединения*

5B | Economy

6A | Economy

6B | Economy

(12 строк)

Если подвести итог, то можно упрощенно объяснить механизм построения соедине-ния следующим образом.

Сначала формируются все попарные комбинации строк из обеих таблиц, т. е. декар-тово произведение множеств строк этих таблиц. Эти комбинированные строки вклю-чают 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 значе-ния 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 мо-жем взять значение атрибута «Модель самолета», чтобы включить ее в итоговый вы-вод данных.

На практике описанный механизм не реализуется буквально. Специальная подсисте-ма PostgreSQL, называемая планировщиком, строит план выполнения запроса, кото-рый 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 исполь-зования 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;**

* этом варианте 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.

155

*Глава 6. Запросы*

Изучая язык SQL вообще и способы выполнения соединений в частности, нужно иметь в виду, что *результатом любых реляционных операций над отношениями (таб-лицами, представлениями) также является отношение*. Поэтому такие операции мож-но произвольно комбинировать друг с другом.

* соединении одна и та же таблица может участвовать дважды, т. е. формировать **соединение таблицы с самой собой**. В качестве примера рассмотрим запрос длясоздания 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;**

156

*6.2. Соединения*

* этом представлении используется не только таблица «Рейсы» (flights), но также и таблица «Аэропорты» (airports). Причем она используется, условно говоря, два-жды. Поясним, что мы имеем в виду.

Как вы уже знаете из главы 3, при соединении двух таблиц в результирующую выбор-ку попадают те комбинации строк из первой и второй таблиц, которые удовлетво-ряют 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 двух таблиц PostgreSQL будет пытаться для каж-дой строки из таблицы flights найти такую строку в таблице airports, в кото-рой 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 дан-ных не учитывается.

Таким образом, приходим к выводу о том, что каждую строку из таблицы «Рейсы» необходимо соединять с двумя *различными* строками из таблицы «Аэропорты»: ведь аэропорт вылета и аэропорт назначения — это *различные* аэропорты. Однако при од-нократном включении таблицы «Аэропорты» в предложение FROM сделать это невоз-можно, поэтому поступают так: к таблице airports в предложении FROM обращают-ся дважды, как будто это две копии одной и той же таблицы.

Конечно, на самом деле никаких копий не создается. Просто в результате поиск строк

* ней будет производиться дважды: один раз для атрибута departure\_airport, а второй раз — для атрибута arrival\_airport. Но поскольку необходимо обес-печить 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 сопоставле-ния с атрибутами departure\_airport и arrival\_airport.

157

*Глава 6. Запросы*

Рассмотрев этот пример, вновь обратимся к соединениям такого типа и покажем три способа выполнения **соединения таблицы с самой собой**, отличающиеся синтак-сически, 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

* 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 предло-жении 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 каж-дой строкой второй таблицы, т. е. формирует **декартово произведение** таблиц — все попарные комбинации строк из двух таблиц. Затем СУБД отбрасывает те комбини-рованные 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 нера-венства значений атрибутов**. Тем самым мы перенесли условие отбора результи-рующих строк из предложения WHERE в предложение FROM.

|  |  |  |
| --- | --- | --- |
| **SELECT count( \*** | **)** |  |
| **FROM airports** | **a1** |  |
| **JOIN airports** | **a2** | **ON a1.city <> a2.city;** |

count

-------

10704

(1 строка)

158

*6.2. Соединения*

Третий вариант предусматривает **явное использование декартова произведения** **таблиц**. Для этого служит предложениеCROSS JOIN. Лишние строки, как и в первомварианте, отсеиваем с помощью предложения WHERE:

**SELECT count( \* )**

**FROM airports a1 CROSS JOIN airports a2**

**WHERE a1.city <> a2.city;**

count

-------

10704

(1 строка)

* точки зрения СУБД эти три варианта эквивалентны и отличаются лишь синтакси-сом. Для них PostgreSQL выберет один и тот же план (порядок) выполнения запроса.

Теперь обратимся к так называемым **внешним соединениям**. Зададимся вопросом: сколько маршрутов обслуживают самолеты каждого типа? Если не требовать вывода наименований моделей самолетов, тогда всю необходимую информацию можно по-лучить из материализованного представления «Маршруты» (routes). Но мы все же будем выводить и наименования моделей, поэтому обратимся также к таблице «Са-молеты» (aircrafts). Соединим эти таблицы на основе атрибута aircraft\_code, сгруппируем строки и просто воспользуемся функцией count. В этом запросе внеш-нее 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)

159

*Глава 6. Запросы*

Обратите внимание, что таблица «Самолеты» содержит 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)

* данном запросе используется **левое внешнее соединение** — об этом говорит пред-ложение LEFT OUTER JOIN.
* 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 ре-зультирующая 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 относитель-но предложения LEFT OUTER JOIN) не находится ни одной соответствующей строки

160

*6.2. Соединения*

* правой таблице, тогда в результирующую строку вместо значений столбцов пра-вой таблицы будут помещены значения NULL. Получается, что для строки из табли-цы 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 стол-бец 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 бу-дет 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 кото-рых значение в этом столбце определено (не равно NULL).

Кроме левого внешнего соединения существует также и **правое внешнее соедине-ние** —RIGHT OUTER JOIN.

* этом случае в качестве базовой выбирается таблица, имя которой указано справа от предложения RIGHT OUTER JOIN, а механизм получения результирующих строк в слу-чае, когда для строки базовой таблицы не находится пары во второй таблице, точно такой же, как и для левого внешнего соединения. Как сказано в документации, пра-вое 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 выше-приведенном запросе мы написали

**SELECT a.aircraft\_code AS a\_code,**

**a.model,**

**r.aircraft\_code AS r\_code,**

**...**

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,**

**...**

161

*Глава 6. Запросы*

Комбинацией этих двух видов внешних соединений является **полное внешнее со-единение** —FULL OUTER JOIN.

* этом случае в выборку включаются строки из левой таблицы, для которых не на-шлось соответствующих строк в правой таблице, и строки из правой таблицы, для которых не нашлось соответствующих строк в левой таблице.
* практической работе при выполнении выборок зачастую выполняются **многотаб-личные запросы**, включающие три таблицы и более. В качестве примера рассмот-рим такую задачу: определить число пассажиров, не пришедших на регистрацию би-летов 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 надо вспомнить, что таблица «Посадочные талоны» (boarding\_passes) связана с таблицей «Перелеты» (ticket\_flights) по внешне-му ключу, а тип связи — 1:1, т. е. каждой строке из таблицы ticket\_flights соот-ветствует *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 регистра-цию 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 стро-ки, соответствующей строке из таблицы ticket\_flights.

162

*6.2. Соединения*

* предложении WHERE второе условие — b.flight\_id IS NULL. Оно и позволяет вы-явить те комбинированные строки, в которых столбцам таблицы boarding\_passes были назначены значения NULL из-за того, что в ней не нашлось строки, для ко-торой выполнялось бы условие t.ticket\_no = b.ticket\_no AND t.flight\_id = b.flight\_id. Конечно, для проверки на NULL мы могли использовать любой стол-бец таблицы boarding\_passes, а не только b.flight\_id.

При формировании соединений подключение таблиц выполняется слева направо,

* 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 описан-ный выше логический порядок соединения таблиц, т. е. взгляд с позиции програм-миста, пишущего запрос, и тот фактический порядок выполнения запроса, который будет сформирован планировщиком. Они могут различаться. Подробно о планах вы-полнения 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 регистра-ции на рейс ему выдали посадочный талон на то место в салоне самолета, где класс обслуживания — Economy. Необходимо выявить все случаи несовпадения классов об-служивания.

Сведения о классе обслуживания, который пассажир выбрал при бронировании би-лета, содержатся в таблице «Перелеты» (ticket\_flights). Однако в таблице «По-садочные талоны» (boarding\_passes), которая «отвечает» за посадку на рейс, све-дений 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 моде-ли 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 ка-честве, 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.

163

*Глава 6. Запросы*

* результате получим запрос, включающий пять таблиц. Он не выдаст ни одной стро-ки, значит, пассажиров, получивших неправильный класс обслуживания, не было.

**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;**

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, и теперь он выдаст одну строку.

* предложении FROM можно использовать виртуальные таблицы, сформированные с помощью **ключевого слова** **VALUES**. Предположим, что для выработки финансовой стратегии нашей авиакомпании требуется распределение количества бронирований по диапазонам сумм с шагом в 100 тысяч рублей. Максимальная сумма в одном бро-нировании составляет 1 204 500 рублей. Учтем это при формировании диапазонов.

Виртуальной таблице, создаваемой с помощью ключевого слова VALUES, присваива-ют имя с помощью ключевого слова AS. После имени в круглых скобках приводится список имен столбцов этой таблицы.

164

*6.2. Соединения*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **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;**

* этом запросе мы использовали внешнее соединение. Сделано это для того, чтобы в случаях, когда в каком-то диапазоне не окажется ни одного бронирования, резуль-тирующая строка выборки все же была бы сформирована. А правое соединение было выбрано только потому, что в качестве первой, базовой, таблицы мы выбрали таб-лицу «Бронирования» (bookings), но именно в ней может не оказаться ни одной строки для соединения с какой-либо строкой виртуальной таблицы. А все строки вир-туальной таблицы, стоящей справа от предложения RIGHT OUTER JOIN, должны быть обязательно представлены в выборке: это позволит сразу увидеть «пустые» диапазо-ны, если они будут.

Можно было использовать и левое внешнее соединение, поменяв 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)

165

*Глава 6. Запросы*

Обратите внимание, что для диапазона от 1 100 до 1 200 тысяч рублей значение счет-чика бронирований равно нулю. Если бы мы не использовали внешнее соединение, то эта строка вообще не попала бы в выборку. Конечно, информация была бы полу-чена та же самая, но воспринимать ее было бы сложнее.

* + команде SELECT предусмотрены средства для выполнения операций с выборками, как с множествами, а именно:

– UNION для вычисления объединения множеств строк из двух выборок;

– INTERSECT для вычисления пересечения множеств строк из двух выборок;

– EXCEPT 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 «Маршру-ты» (routes).

Начнем с операции **объединения множеств строк —** **UNION**. Строка включается

* 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

...

(87 строк)

Рассмотрим операцию **пересечения множеств строк —** **INTERSECT**. Строка вклю-чается 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.

166

*6.2. Соединения*

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

...

(15 строк)

* завершение рассмотрим операцию **разности множеств строк —** **EXCEPT**. Строка включается в итоговое множество (выборку), если она присутствует в первом мно-жестве (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

...

(7 строк)

Конечно, при выполнении этих операций можно соединять не только две таблицы, но

* большее их число. При этом нужно либо учитывать приоритеты выполнения опера-ций, либо использовать скобки. Согласно документации INTERSECT связывает свои подзапросы сильнее, чем UNION, а EXCEPT связывает свои подзапросы так же сильно, как UNION.

167

*Глава 6. Запросы*

**6.3. Агрегирование и группировка**

Среди множества функций, имеющихся в PostgreSQL, важное место занимают агре-гатные 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 таб-лицы «Бронирования».

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 предложе-ния 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 неде-лю или семь раз, он учитывается только однократно. Воспользуемся материализо-ванным представлением «Маршруты».

168

*6.3. Агрегирование и группировка*

**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 |
| ... |  |  |

* 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». 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)

* этом запросе используется функция array\_length, возвращающая количество элементов в указанном измерении массива. Поскольку массив одномерный, то вто-рым параметром функции будет число 1 — первое измерение.

При выполнении выборок можно с помощью условий, заданных в предложении WHERE, сузить множество выбираемых строк. Аналогичная возможность существу-ет и при выполнении группировок: можно включить в результирующее множество не все строки, а лишь те, которые удовлетворяют некоторому условию. Это условие

169

*Глава 6. Запросы*

можно задать в **предложении** **HAVING**. Важно помнить, что предложение WHERE ра-ботает с отдельными строками еще до выполнения группировки с помощью GROUP BY, а предложение HAVING — уже после выполнения группировки.

* качестве примера приведем такой запрос: определить, сколько существует марш-рутов 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)

* 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 функ-ции (window functions)**, технология использования которых описана в документа-ции в разделе 3.5 «Оконные функции». Эти функции предоставляют возможность производить вычисления на множестве строк, логически связанных с текущей стро-кой, 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.

170

*6.3. Агрегирование и группировка*

Предположим, что руководство нашей компании хочет усовершенствовать тарифную политику и с этой целью просит нас предоставить сведения о распределении коли-чества проданных билетов на некоторые рейсы во времени. Количество проданных билетов должно выводиться в виде накопленного показателя, суммирование должно производиться в пределах каждого календарного месяца.

Более детально, в столбцах book\_ref и book\_date выборки должны приводиться но-мер и время бронирования соответственно. В столбцах month и day должны указы-ваться порядковый номер месяца и день этого месяца. Столбец count должен содер-жать 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 мно-жество строк, у которых даты продажи билета (т. е. даты бронирования) относятся к одному и тому же месяцу. В результате в полученной выборке, пример которой при-веден ниже, будет сформировано два раздела.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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)

Здесь для примера был выбран рейс с идентификатором 1.

171

*Глава 6. Запросы*

Понятие оконного кадра (window frame) является важным, поскольку многие окон-ные функции работают не со всеми строками раздела, а только с теми, которые обра-зуют оконный кадр текущей строки. Если строки в разделе не упорядочены, то окон-ным 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 *по-сле* 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 име-ет следующее объяснение. В рамках одного бронирования с номером 854C4C были проданы сразу три билета на этот рейс, поэтому в этих трех строках значения в столб-це book\_date одинаковые. Строки в выборке упорядочены по значениям столбца book\_date. Таким образом, для каждой из этих трех строк, т. е. для третьей, четвер-той и пятой, значения критерия сортировки одинаковые, поэтому оконным кадром для каждой из них будут являться первые пять строк первого раздела выборки. Под-счет 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.

* приведенной выборке отражены также и случаи одновременного бронирования двух билетов на данный рейс. Вы можете найти соответствующие строки самосто-ятельно.

Теперь посмотрим, с помощью какого запроса был получен этот результат, и на его примере объясним синтаксические конструкции, используемые для работы с окон-ными 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;**

172

*6.3. Агрегирование и группировка*

Рассмотрим конструкцию, предназначенную для вызова оконной функции:

**count( \* ) OVER (**

**PARTITION BY date\_trunc( 'month', b.book\_date )**

**ORDER BY b.book\_date**

* + **AS count**
* 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 раз-биения 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 предложе-ния 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 теку-щей 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) то-же 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 группи-ровки строк, а работают на уровне отдельных (несгруппированных) строк. Однако ес-ли в запросе присутствуют предложения GROUP BY и HAVING, тогда оконные функции вызываются уже *после* них. В таком случае оконные функции будут работать со стро-ками, являющимися результатом группировки.

Рассмотрим еще один пример. Покажем, как с помощью оконной функции rank мож-но проранжировать аэропорты в пределах каждого часового пояса на основе их гео-графической широты. Причем будем присваивать более высокий ранг тому аэропор-ту, который находится севернее.

173

*Глава 6. Запросы*

**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;**

* этом запросе в предложении OVER ( PARTITION BY timezone ... ) указывает-ся, что строки относятся к одному разделу на основе совпадения значений в столбце timezone. Обратите внимание, что хотя в предложении OVER задан порядок сортиров-ки, 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 |
| ... |  |  |  |  |  |  |  |
| 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 гео-графической широтой и широтой, на которой находится самый северный аэропорт

* этом же часовом поясе. Поскольку в запросе используются три конструкции с окон-ными функциями и при этом способ формирования разделов и порядок сортировки строк в разделах один и тот же, то вводится предложение WINDOW. Оно позволяет со-здать определение раздела, а затем ссылаться на него при вызове оконных функций. Самый северный аэропорт в каждом часовом поясе, т. е. самая первая строка в каж-дом 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.

174

*6.3. Агрегирование и группировка*

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 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

...

Более подробно использование оконных функций описано в документации. Мы ре-комендуем начать с раздела 3.5 «Оконные функции», в котором приводятся приме-ры их использования. В разделе 9.21 «Оконные функции» приводятся описания всех оконных функций, предлагаемых PostgreSQL. В разделе 4.2.8 «Вызовы оконных функ-ций» 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.

175

*Глава 6. Запросы*

**6.4. Подзапросы**

Прежде чем приступить к рассмотрению столь сложной темы, как подзапросы, опи-шем, как в общем случае работает команда SELECT. Согласно описанию этой коман-ды, приведенному в документации, 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 таб-лицы, но также и виртуальные таблицы, создаваемые с помощью ключевого слова VALUES. Если таблиц больше одной, то формируется декартово произве-дение из множеств их строк. Например, в случае двух таблиц будут сформиро-ваны попарные комбинации каждой строки из одной таблицы с каждой строкой из другой таблицы. При этом в комбинированных строках сохраняются все ат-рибуты из каждой исходной таблицы.
2. Если в команде присутствует условие WHERE, то из полученного декартова про-изведения исключаются строки, которые этому условию не соответствуют. Та-ким образом, первоначальное множество строк, сформированное без всяких условий, сужается.
3. Если присутствует предложение GROUP BY, то результирующие строки группи-руются на основе совпадения значений одного или нескольких атрибутов, а за-тем вычисляются значения агрегатных функций. Если присутствует предложе-ние HAVING, то оно отфильтровывает результирующие строки (группы), не удо-влетворяющие критерию.
4. Ключевое слово SELECT присутствует всегда. Но в списке выражений, идущих после него, могут быть не только простые имена атрибутов, но и их комбина-ции, 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, ат-рибуты, на основе которых формируются условия в предложении WHERE, могут отсутствовать в списке выражений после ключевого слова SELECT. Предложе-ние SELECT DISTINCT удаляет дубликаты строк.
5. Если присутствует предложение ORDER BY, то результирующие строки сорти-руются 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.

176

*6.4. Подзапросы*

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 раздела — подзапросам.

Предположим, что сотрудникам аналитического отдела потребовалось провести ста-тистическое исследование финансовых результатов работы авиакомпании. В каче-стве первого шага они решили подсчитать количество операций бронирования, в ко-торых 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)

* приведенном запросе присутствует два предложения SELECT, но при этом толь-ко одно из них является главным в этом запросе, а другое представляет собой **под-запрос**. 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.
* приведенном примере в предложении WHERE используется так называемый **ска-лярный подзапрос**. Это означает, что в результате его выполнения возвращаетсятолько одно скалярное значение (один столбец и одна строка), с которым можно срав-нивать другие скалярные значения.

Если подзапрос выдает множество скалярных значений (или даже только одно), мож-но использовать такой **подзапрос в предикате** **IN**. Этот предикат позволяет органи-зовать проверку на предмет принадлежности какого-либо значения определенному множеству значений.

* качестве примера давайте выясним, какие маршруты существуют между городами часового пояса Asia/Krasnoyarsk. Подзапрос будет выдавать список городов из это-го 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 обра-ботке каждой строки из таблицы routes во внешнем запросе. Повторного выполне-ния подзапроса не требуется, т. к. его результат не зависит от значений, хранящихся в таблице routes. Такие подзапросы называются **некоррелированными**.

177

*Глава 6. Запросы*

**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.

178

*6.4. Подзапросы*

Иногда возникают ситуации, когда от подзапроса требуется лишь установить сам факт наличия или отсутствия строк в конкретной таблице, удовлетворяющих опре-деленному 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;**

* этом запросе мы не можем ограничиться только лишь материализованным пред-ставлением «Маршруты» (routes), поскольку в нем представлены лишь *существу-ющие* 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 на-личия или отсутствия строк, соответствующих критерию отбора, то в документации рекомендуется вместо списка столбцов (или символа « ») в предложении SELECT де-лать 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.

* еще одна важная деталь. В представленном запросе мы использовали так называ-емый **коррелированный (связанный) подзапрос**. В подзапросах такого типа при-сутствует ссылка (ссылки) на таблицу из внешнего запроса, как здесь:

**WHERE ...**

**AND r.arrival\_city = a.city**

179

*Глава 6. Запросы*

* теории это означает, что подзапрос выполняется не один раз для всего внешне-го 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 ва-риант 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;**

180

*6.4. Подзапросы*

Обратите внимание, что в этом запросе мы использовали коррелированные под-запросы. Все они ссылаются на столбец таблицы «Самолеты» (aircrafts), кото-рая обрабатывается во внешнем запросе. Для каждой обрабатываемой строки таб-лицы aircrafts подсчитывается число строк в таблице seats, в которых атрибут aircraft\_code имеет такое же значение, что и в строке таблицы aircrafts. Под-запросы 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;**

181

*Глава 6. Запросы*

Подзапрос формирует временную таблицу в таком виде:

|  |  |  |  |
| --- | --- | --- | --- |
| model | | fare\_conditions | | num | |
| --------------------- | +----------------- | + | ----- |
| Airbus A319-100 | | Business | | | 20 |
| Airbus A319-100 | | Economy | | | 96 |
| ... |  |  |  |
| Sukhoi SuperJet-100 | Business | | | | 12 |
| Sukhoi SuperJet-100 | Economy | | | | 85 |

(17 строк)

* 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 воз-вращает не числовое значение, а строку символов, составленную из значений атрибу-тов, 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)

* качестве еще одного примера использования подзапроса в предложении FROM ре-шим 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;**

182

*6.4. Подзапросы*

Благодаря использованию предложения HAVING, подзапрос выбирает города, в кото-рых имеется более одного аэропорта, и формирует временную таблицу в следующем виде:

|  |  |  |
| --- | --- | --- |
| city | | count | |
| ----------- | + | ------- |
| Ульяновск | | | 2 |
| Москва | | | 3 |

(2 строки)

* в главном запросе выполняется соединение временной таблицы с таблицей «Аэро-порты» (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 та-кую 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)

183

*Глава 6. Запросы*

* сложных запросах могут использоваться **вложенные подзапросы**. Это означает, что один подзапрос находится внутри другого. Давайте в качестве примера рассмот-рим такую ситуацию: руководство авиакомпании хочет выяснить степень заполне-ния самолетов на всех рейсах, ведь отправлять полупустые самолеты не очень выгод-но. 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 само-лете. Этот подзапрос — коррелированный, т. к. он выполняется для каждой строки,

184

*6.4. Подзапросы*

обрабатываемой во внешнем подзапросе, т. е. для каждой модели самолета. Для под-счета числа проданных билетов мы использовали соединение представления «Рей-сы» (flights\_v) с таблицей «Перелеты» (ticket\_flights) с последующей группи-ровкой строк и вызовом функции count. Конечно, можно было бы вместо такого ре-шения использовать еще один коррелированный подзапрос:

* **SELECT count( tf.ticket\_no ) FROM ticket\_flights tf**

**WHERE tf.flight\_id = f.flight\_id**

**) AS fact\_passengers**

* таком случае уже не потребовалось бы соединять представление flights\_v с таб-лицей ticket\_flights и группировать строки, достаточно было бы сделать:

**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 вы-числение степени заполнения самолета пассажирами, а также сортирует результиру-ющие 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 |
| ... |  |  |

185

*Глава 6. Запросы*

Рассмотренный сложный запрос можно сделать более наглядным за счет выделения подзапроса в отдельную конструкцию, которая называется **общее табличное выра-жение (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 присутство-вать больше одного подзапроса. Каждый подзапрос формирует временную таблицу

* 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 запро-се обращаться к временной таблице ts так, как если бы она существовала постоянно.

186

*6.4. Подзапросы*

Но важно учитывать, что временная таблица, создаваемая в общем табличном выра-жении, существует только во время выполнения запроса.

* этой главе мы уже решали задачу распределения сумм бронирований по диапазо-нам с шагом в 100 тысяч рублей. Тогда мы использовали предложение VALUES для формирования виртуальной таблицы. Можно решить эту задачу более рациональ-ным способом с использованием конструкции WITH ... AS (...).

Сначала покажем, как можно сформировать диапазоны сумм бронирований с помо-щью **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;**

* + 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 записывает-ся во временную область памяти;

– затем к этой временной области памяти применяется запрос

**SELECT min\_sum + 100000, max\_sum + 100000**

**...**

и в результате его выполнения формируется только одна строка, поскольку в ис-ходном 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 коман-ды SELECT, опять применяется эта же команда и т. д.;

– работа завершится, когда перестанет выполняться условие

**max\_sum < ( SELECT max( total\_amount ) FROM bookings )**

187

*Глава 6. Запросы*

Важную роль в этом процессе играет предложение UNION ALL, благодаря которо-му происходит объединение сформированных строк в единую таблицу. Поскольку

* нашем примере в рекурсивном алгоритме участвует только одна строка, то строк-дубликатов не возникает, поэтому мы используем не UNION, а UNION ALL. При ис-пользовании предложения UNION выполняется устранение строк-дубликатов, но в данном случае необходимости в выполнении этой операции нет, следовательно, це-лесообразно использовать именно UNION ALL.

Получим такую таблицу:

min\_sum | max\_sum

|  |  |  |
| --- | --- | --- |
| --------- | + | -------- |
| 0 | | | 100000 |
| 100000 | | | 200000 |
| 200000 | | | 300000 |
| ... |  |  |
| 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;**

188

*6.4. Подзапросы*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 1 100 до 1 200 тысяч рублей значение числа бронирований равно нулю. Для того чтобы была выведена строка с нулевым значе-нием столбца count, мы использовали внешнее соединение.

* заключение рассмотрим команду для создания материализованного представле-ния «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[ ] |

189

*Глава 6. Запросы*

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**

**)**

190

*6.4. Подзапросы*

**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, содер-жит в предложении FROM подзапрос, формирующий временную таблицу f2. А этот подзапрос, в свою очередь, также содержит в предложении FROM подзапрос, форми-рующий временную таблицу f1. Таким образом, в этой команде используется вло-женный 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: по-недельник — 1, воскресенье — 7. Поскольку номер дня недели представлен в виде символьной строки, он преобразуется в тип данных integer. Таким образом, вло-женный подзапрос вычисляет плановую длительность полета (столбец duration)

* извлекает номер дня недели из даты/времени вылета по расписанию (столбец days\_of\_week).

Подзапрос следующего, более высокого уровня, получив результат вложенного под-запроса, 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 длитель-ный период. Поэтому рейс, который отправляется, скажем, по вторникам, появится

* этом расписании несколько раз, следовательно, день недели с номером 2 также по-явится в столбце days\_of\_week для этого номера рейса несколько раз. В результате,

191

*Глава 6. Запросы*

если не прибегнуть к группировке по этому столбцу, то при формировании масси-ва дней недели в этом массиве будут многократные вхождения каждого дня недели, когда этот рейс летает. В этом подзапросе присутствует и предложение ORDER BY,

* которое включен столбец days\_of\_week. Это необходимо для того, чтобы агре-гатная 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

* 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 таб-лицей «Аэропорты» (airports), причем дважды. Это нужно потому, что в таб-лице f3 есть столбец f3.departure\_airport (аэропорт отправления) и столбец f3.arrival\_airport (аэропорт прибытия), для каждого из них нужно выбрать на-именование аэропорта и наименование города из таблицы airports. О том, как нужно рассуждать при двукратном использовании одной и той же таблицы в соеди-нении, 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 после-довательности, однако ничего не сказано насчет правил обработки пробелов.

* таблице «Билеты» (tickets) столбец passenger\_name содержит имя и фами-лию 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** | **'%** | **%%';** |

192

*Контрольные вопросы и задания*

1. Этот запрос выбирает из таблицы «Билеты» (tickets) всех пассажиров с име-нами, состоящими из трех букв (в шаблоне присутствуют 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.

1. В разделе документации 9.7.2 «Регулярные выражения SIMILAR TO» рассмат-ривается оператор SIMILAR TO. Он работает аналогично оператору LIKE, но использует шаблоны, соответствующие определению регулярных выражений, приведенному в стандарте SQL. Регулярные выражения SQL представляют со-бой комбинацию синтаксиса LIKE с синтаксисом обычных регулярных выраже-ний. Самостоятельно ознакомьтесь с оператором SIMILAR TO.
2. В разделе документации 9.2 «Функция и операторы сравнения» представлены различные предикаты сравнения, кроме предиката BETWEEN, рассмотренного в этой главе. Самостоятельно ознакомьтесь с ними.
3. В разделе документации 9.17 «Условные выражения» представлены услов-ные выражения, которые поддерживаются в PostgreSQL. В тексте главы бы-ла рассмотрена конструкция CASE. Самостоятельно ознакомьтесь с функциями

COALESCE, NULLIF, GREATEST и LEAST.

1. Выясните, на каких маршрутах используются самолеты компании Boeing. В вы-борке 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).

1. Самые крупные самолеты в нашей авиакомпании — это Boeing 777-300. Выяс-нить, 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;**

193

*Глава 6. Запросы*

|  |  |  |
| --- | --- | --- |
| departure\_city | | | arrival\_city |
| ---------------- | + | -------------- |
| Екатеринбург | | | Москва |
| Москва | | | Екатеринбург |
| Москва | | | Новосибирск |
| Москва | | | Пермь |
| Москва | | | Сочи |
| Новосибирск | | | Москва |
| Пермь | | | Москва |
| Сочи | | | Москва |

(8 строк)

* сожалению, в этой выборке информация дублируется. Пары городов приведе-ны 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)

1. В тексте главы мы рассматривали различные примеры использования левого и правого внешних соединений: LEFT OUTER JOIN и RIGHT OUTER JOIN. Напи-шите запрос, в котором использовалось бы полное внешнее соединение — FULL OUTER JOIN.
2. Для ответа на вопрос, сколько рейсов выполняется из Москвы в Санкт-Петер-бург, можно написать совсем простой запрос:

**SELECT count( \* )**

**FROM routes**

**WHERE departure\_city = 'Москва'**

**AND arrival\_city = 'Санкт-Петербург';**

count

-------

12

(1 строка)

194

*Контрольные вопросы и задания*

А с помощью какого запроса можно получить результат в таком виде?

departure\_city | arrival\_city | count

----------------+-----------------+-------

Москва | Санкт-Петербург | 12

(1 строка)

1. Выяснить, сколько различных рейсов выполняется из каждого города, без уче-та 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 |
| ... |  |  |
| Благовещенск | | | 1 |
| Братск | | | 1 |

(101 строка)

Модифицируйте этот запрос так, чтобы он выводил число направлений, по ко-торым 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.

1. В материализованном представлении «Маршруты» (routes) имеется столбец days\_of\_week, который содержит списки (массивы) номеров дней недели, ко-гда выполняется каждый рейс.

Для оптимизации расписания вылетов из Москвы нужно выявить пять горо-дов, в которые из столицы отправляется наибольшее число ежедневных рейсов (маршрутов). Строки в выборке следует расположить в убывающем порядке чис-ла выполняемых рейсов.

Указание: воспользуйтесь функцией array\_length.

195

*Глава 6. Запросы*

12.\* Предположим, что служба материального снабжения нашей авиакомпании за-просила 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[];**

* этом запросе используется оператор @>, который проверяет, содержатся ли все элементы массива, стоящего справа от него, в том массиве, который нахо-дится 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 номером дня недели 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;**

196

*Контрольные вопросы и задания*

|  |  |  |
| --- | --- | --- |
| 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 указа-ны в главе 4. Однако можно воспользоваться и предметным указателем (Index), ссылка на который находится в самом низу оглавления документации.

* 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 недолжно 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». Функ-ция может принимать любое число параметров-массивов, а возвращает набор

197

*Глава 6. Запросы*

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. Предложение WITH ORDINALITY позволяет в на-шем примере избавиться от массива целых чисел, обозначающих дни неде-ли, поскольку автоматически формируется столбец целых чисел, нумерую-щих строки результирующего набора. По умолчанию этот столбец называется ordinality. Это имя можно использовать в запросе. Самостоятельно модифи-цируйте запрос с применением предложения WITH ORDINALITY.

1. 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)

198

*Контрольные вопросы и задания*

* 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 мини-мальная 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.

1. Предположим, что маркетологи нашей авиакомпании хотят знать, как часто встречаются различные имена среди пассажиров? Получить распределение ча-стот 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 ча-стот фамилий пассажиров.

Подробные сведения о других функциях для работы со строковыми данными приведены в документации в разделе 9.4 «Строковые функции и операторы».

199

*Глава 6. Запросы*

15.\* В тексте главы были кратко рассмотрены оконные функции. Самостоятельно прочитайте разделы 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 документа-ции 4.2.7 «Агрегатные выражения». Напишите запрос с использованием пред-ложения FILTER с агрегатной функцией.

1. В тексте главы в разделе 6.4 мы рассмотрели два способа получения ответа на вопрос: как распределяются места с разными классами обслуживания в самоле-тах 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) |  |  |  |  |  |

1. В разделе 6.2 мы находили ответ на вопрос: сколько маршрутов обслуживают са-молеты 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)

200

*Контрольные вопросы и задания*

19.\* В разделе 6.4 мы использовали рекурсивный алгоритм в общем табличном вы-ражении. Изучите этот пример, чтобы лучше понять работу рекурсивного алго-ритма:

**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)

201

*Глава 6. Запросы*

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 про-цесс порождения новых строк.

При рекурсивном выполнении запроса

**SELECT min\_sum + 100000, max\_sum + 100000 FROM ranges**

**WHERE max\_sum < ( SELECT max( total\_amount ) FROM bookings )**

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 от-клонена условием WHERE. Работа алгоритма завершается.

**Задание 1.** Модифицируйте запрос, добавив в него столбецlevel(можно на-звать его и iteration). Этот столбец должен содержать номер текущей итера-ции, 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 заменитеUNION ALLнаUNIONивыполните запрос. Сравните этот результат с предыдущим, когда мы исполь-зовали UNION ALL.

202

*Контрольные вопросы и задания*

20.\* В тексте главы есть такой запрос, вычисляющий распределение сумм брониро-ваний по диапазонам в 100 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 выра-жение b.\*, а не просто \*? Что изменится, если оставить только \*, и почему?

1. 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 приведенном ниже запросе нужное ключевое сло-во — UNION, INTERSECT или EXCEPT — и обоснуйте ваше решение.

203

*Глава 6. Запросы*

**SELECT city**

**FROM airports**

**WHERE city <> 'Москва'**

**?**

**SELECT arrival\_city**

**FROM routes**

**WHERE departure\_city = 'Москва'**

**ORDER BY city;**

1. 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 пред-ложении SELECT или можно написать просто

**SELECT city FROM airports**

Сначала попробуйте дать ответ теоретически, а потом проверьте вашу гипотезу на компьютере.

1. Предположим, что департамент развития нашей авиакомпании задался вопро-сом: 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 *аэро-портами*. Здесь мы используем соединение таблицы с самой собой на основенеравенства 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;**

204

*Контрольные вопросы и задания*

count

-------

10100

(1 строка)

**Задание.** Перепишите этот запрос с общим табличным выражением.

1. 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 подзапро-сов». Самостоятельно ознакомьтесь с этими предикатами и напишите несколь-ко запросов с их применением.

Предикаты ANY и ALL имеют некоторую связь с предикатом IN. В частности, использование IN эквивалентно использованию конструкции = ANY, а исполь-зование NOT IN эквивалентно использованию конструкции <> ALL.

Пример двух эквивалентных запросов, выбирающих аэропорты в часовых поя-сах 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 ( ... )**

205

*Глава 6. Запросы*

25.\* При планировании новых маршрутов и оценке экономической эффективности уже существующих может потребоваться информация о том, какова усреднен-ная 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\_pass,** |  |  |  |
|  | **sum( ts.total\_seats ) AS sum\_seats,** | | | |  |  |  |
|  | **round( sum( ts.fact\_passengers )::numeric /** | | | | |  |  |
|  | **sum( ts.total\_seats** | | **)::numeric, 2 ) AS frac** | | | |  |
| **FROM tickets\_seats ts** | | |  |  |  |  |  |
| **GROUP** | **BY ts.departure\_city, ts.arrival\_city** | | | |  |  |  |
| **ORDER** | **BY ts.departure\_city;** | |  |  |  |  |  |
| departure\_city | | | arrival\_city | | | sum\_pass | | sum\_seats | | | frac |
| ---------------- | + | ----------------- | + | ---------- | +----------- | | +------ |
| Абакан | | | Tomsk | | | 258 | | | 360 | | 0.72 |
| Абакан | | | Novosibirsk | | | 217 | | | 348 | | 0.62 |
| Абакан | | | Moscow | | | 466 | | | 1044 | | 0.45 |
| ... |  |  |  |  |  |  |  |
| Якутск | | | Санкт-Петербург | | | 352 | | | 3596 | | 0.10 |

(361 строка)

Для того чтобы лучше уяснить, как работает запрос в целом, вычлените из него отдельные подзапросы и выполните их, посмотрите, что они выводят.

206

*Контрольные вопросы и задания*

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 дан-ные, но с учетом классов обслуживания, т. е. Business, Comfort и Economy.

26.\* Предположим, что некая контролирующая организация потребовала информа-цию о размещении пассажиров одного из рейсов Кемерово — Москва в салоне самолета. Для определенности выберем конкретный рейс из тех рейсов, кото-рые уже прибыли на момент времени, соответствующий текущему моменту. Текущий момент времени в базе данных «Авиаперевозки» определяется с по-мощью 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 назначе-ны 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;**

207

*Глава 6. Запросы*

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) |  |  |  |

208

*Контрольные вопросы и задания*

Получим список мест в салоне самолета и пассажиров, которые сидели на этих местах. При этом незанятые места также должны быть выведены (поэтому ис-пользуем левое внешнее соединение 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 | |
| ... | |  |  |
| 8F |  | | |  |
| 9A |  | | MAKSIM CHERNOV | |
| 9C |  | | |  |
| 9D |  | | LYUDMILA IVANOVA | |
| 9E |  | | |  |
| 9F |  | | SOFIYA KULIKOVA | |
| (97 | строк) | |  |

Предположим, что нас попросили отсортировать места в порядке их расположе-ния в салоне самолета и вывести также адреса электронной почты пассажиров (у кого они были указаны при бронировании). Для выполнения второго требова-ния воспользуемся столбцом contact\_data. В нем содержатся JSON-объекты,

209

*Глава 6. Запросы*

содержащие контактные данные пассажиров. Ряд из них имеет ключ email. Мо-дифицированный 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 | | |  | | |
| ... |  |  |  |
| 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 табличноговыражения и добавьте столбец «Класс обслуживания» (fare\_conditions).

210

**Глава 7**

**Изменение данных**

Эта глава будет посвящена операциям изменения данных: вставке новых строк в таблицы, обновле-нию 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 прово-дить эксперименты, будучи уверенными в том, что данные в постоянных таблицах модифицированы не будут, поэтому все запросы, которые вы выполняли ранее, бу-дут работать так, как и работали.

Итак, создадим две копии таблицы «Самолеты» (aircrafts). Первая таблица-копия предназначена для хранения данных, взятых из таблицы-прототипа, а вторая табли-ца-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 air-crafts не будем, о чем говорит предложение WITH NO DATA. Если бы мы решили скопировать в новую таблицу и все строки, содержащиеся в таблице-прототипе, то-гда 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 накла-дывать ограничения на нее. Однако в нашем случае речь о массовом вводе не идет,

211

*Глава 7. Изменение данных*

поэтому мы начнем с наложения ограничений, а уже потом добавим строки в таб-лицу.

**ALTER TABLE aircrafts\_tmp**

**ADD PRIMARY KEY ( aircraft\_code );**

**ALTER TABLE aircrafts\_tmp**

**ADD UNIQUE ( model );**

Теперь создадим вторую таблицу, и также не будем копировать в нее данные из по-стоянной таблицы aircrafts.

**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 предложе-нием 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 пред-ложение LIKE можно применять для создания не только временных таблиц, но и постоянных.

Поскольку у нас есть журнальная таблица aircrafts\_log, мы можем записывать в нее все операции с таблицей aircrafts\_tmp, т. е. вести историю изменений данных таблицы aircrafts\_tmp.

212

*7.1. Вставка строк в таблицы*

Начнем работу с того, что скопируем в таблицу aircrafts\_tmp все данные из табли-цы aircrafts. Для выполнения не только «полезной» работы, но и ведения журнала изменений мы используем **команду** **INSERT** **с общим табличным выражением**.

Вообще, при классическом подходе для ведения учета изменений, внесенных в таб-лицы, используют триггеры или правила (rules), но их рассмотрение выходит за рам-ки этого пособия. Поэтому наш пример нужно рассматривать как иллюстрацию воз-можностей 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 «по-лезная» работа выполняется в рамках конструкции WITH add\_row AS (...). Здесь строки с помощью команды SELECT выбираются из таблицы aircrafts и вставля-ются в таблицу aircrafts\_tmp. При вставке строк, выбранных из одной таблицы,

* другую таблицу необходимо, чтобы число атрибутов и их типы данных во вставляе-мых строках были согласованы с числом столбцов и их типами данных в целевой таб-лице. Завершается конструкция WITH add\_row AS (...) предложением RETURNING \*, которое просто возвращает внешнему запросу все строки, успешно добавленные
* таблицу aircrafts\_tmp. Конечно же, при этом из таблицы aircrafts\_tmp добав-ленные 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. Этот запрос, в свою очередь, получает данные от временной таблицы add\_row, указанной в предложении FROM. Поскольку в предложении RETURNING мы указали

* качестве возвращаемого значения \*, то будут возвращены все столбцы таблицы aircrafts\_tmp, т. е. той таблицы, в которую строки были *вставлены*. Следователь-но, в команде SELECT внешнего запроса можно ссылаться на имена этих столбцов:

**SELECT add\_row.aircraft\_code, add\_row.model, add\_row.range, ...**

213

*Глава 7. Изменение данных*

Поскольку в таблице aircrafts\_log существует еще два столбца, то для них мы до-полнительно 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,

214

*7.1. Вставка строк в таблицы*

оно предусматривает два варианта действий на выбор программиста. Первый вари-ант — отменять добавление новой строки, для которой имеет место конфликт зна-чений ключевых атрибутов, и при этом не порождать сообщения об ошибке. Второй вариант заключается в замене операции добавления новой строки операцией обнов-ления существующей строки, с которой конфликтует добавляемая строка.

Начнем с первого варианта. Попробуем добавить строку, которая гарантированно бу-дет 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 сообщение о нуле строк относится к таблице aircrafts\_log,

* е. к команде в главном запросе, а не в общем табличном выражении, в котором мы работаем с таблицей aircrafts\_tmp. Проверьте, не была ли добавлена строка в таблицу aircrafts\_tmp.
* том случае, когда в предложении ON CONFLICT не указана дополнительная инфор-мация об именах столбцов или ограничений, по которым предполагается возможный конфликт, проверка выполняется по первичному ключу и по уникальным ключам.

Укажем конкретный столбец для проверки конфликтующих значений. Пусть это бу-дет aircraft\_code, т. е. первичный ключ. Для упрощения команды не будем ис-пользовать общее табличное выражение. Добавляемая строка будет конфликтовать 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 \*;**

215

*Глава 7. Изменение данных*

Получим только такое сообщение:

aircraft\_code | model | range

---------------+-------+-------

(0 строк)

INSERT 0 0

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 выпол-нение требования уникальности по столбцу model. В результате мы получим тради-ционное 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 опе-рация вставки новой строки заменяется операцией обновления существующей стро-ки, с которой и возник конфликт значений столбцов. Для реализации этой возмож-ности служит предложение DO UPDATE.

Давайте модифицируем команду и добавим предложение DO UPDATE. Выберем та-кую политику для работы с таблицей aircrafts\_tmp: если при вставке новой строки имеет место дублирование по атрибутам первичного ключа со строкой, находящей-ся 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.

* качестве примера сделаем так: в добавляемой строке значение атрибута model сде-лаем отличающимся от того, которое уже есть в таблице (вместо Sukhoi SuperJet-100 будет Sukhoi SuperJet), а значение атрибута range оставим без изменений (3000).

Внесем еще одно изменение: вместо имени столбца, образующего первичный ключ,

* помощью предложения ON CONSTRAINT укажем наименование ограничения пер-вичного ключа. Вот так выглядит команда с предложением DO UPDATE:

216

*7.1. Вставка строк в таблицы*

**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

* 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)

* случае конфликта по столбцу aircraft\_code будет обновлена та строка в табли-це aircrafts\_tmp, с которой конфликтовала вновь добавляемая строка. В резуль-тате новая строка добавлена не будет, а будет обновлено значение столбца model в строке, уже находящейся в таблице. А где PostgreSQL возьмет значение для ис-пользования 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 зна-чение — 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 опера-ции 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 появил-ся, 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 копировать данные из таблиц в файлы и на стандартный вывод.

217

*Глава 7. Изменение данных*

* качестве примера ввода данных из файла давайте добавим две строки в табли-цу aircrafts\_tmp. Сначала необходимо подготовить текстовый файл, содержащий новые данные. В этом файле каждая строка соответствует одной строке таблицы. Зна-чения атрибутов разделяются символами табуляции, поэтому пробелы, которые есть в столбце model, можно вводить в файл без каких-либо дополнительных экранирую-щих символов. Заключать строковые значения в одинарные кавычки не нужно, ина-че они также будут введены в таблицу. Завершить файл нужно строкой, содержащей только символы «n.». 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 таб-лице, удалены не были.

При использовании команды COPY выполняются проверки всех ограничений, нало-женных 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.

218

*7.2. Обновление строк в таблицах*

**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 выполня-ется 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 за-просу, в котором выполняется операция INSERT, добавляющая строку в журнальную таблицу. Конечно, если бы строка в таблице aircrafts\_tmp не была успешно обнов-лена, 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 учиты-вать, что главный запрос может получить доступ к обновленным данным *только че-рез временную таблицу*, которую формирует предложениеRETURNING:

**...**

**FROM update\_row ur;**

Можно выполнить выборку из журнальной таблицы aircrafts\_log, чтобы посмот-реть — правда, не очень длинную — историю изменений строки с описанием само-лета Bombardier CRJ-200.

219

*Глава 7. Изменение данных*

**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 видеть динамику

продаж билетов по всем направлениям, а именно: общее число проданных билетов

* + дату/время последнего увеличения их числа для конкретного направления. Создадим временную таблицу tickets\_directions с четырьмя столбцами:

– города отправления и прибытия — departure\_city и 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» 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 уни-кальные 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;** |

220

*7.2. Обновление строк в таблицах*

Поскольку PostgreSQL не требует обязательного создания первичного ключа, то не будем создавать его. Это не помешает нам однозначно идентифицировать строки

* таблице tickets\_directions.

Поскольку в команде ALTER TABLE нет предложения WHERE, в котором было бы усло-вие, 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», однако без внешних ключей. Поэтому мы сможем добавлять в нее строки, не заботясь о добавлении строк в таблицы «Биле-ты» и «Бронирования». Тем не менее первичный ключ все же создадим, чтобы проде-монстрировать, что в случае попытки ввода строк с дубликатными значениями пер-вичного 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 би-лета и увеличивать в таблице tickets\_directions значение счетчика проданных билетов.

**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

221

*Глава 7. Изменение данных*

Этот запрос работает следующим образом. Добавление новой записи о бронирова-нии авиаперелета производится в общем табличном выражении, а наращивание со-ответствующего счетчика — в главном запросе. Поскольку в общем табличном вы-ражении присутствует предложение RETURNING \*, значения атрибутов добавлен-ной 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.

* главном запросе мы обновляем всего два атрибута, причем значение атрибута tickets\_num может увеличиться только на единицу, поскольку мы добавляем од-ну строку в таблицу ticket\_flights\_tmp. Остается выяснить, каким образом мож-но 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 при-бытия, 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 идентифика-тора рейса flight\_id, на который был забронирован билет (перелет). Назначение вложенного подзапроса в том, чтобы в условии WHERE flight\_id = ... не дубли-ровать 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 пер-вого варианта состоит в том, что для определения обновляемой строки в таблице

222

*7.2. Обновление строк в таблицах*

tickets\_directions используется **операция соединения таблиц**. Здесь в глав-ном запросе UPDATE присутствует предложение FROM, однако в этом предложе-нии указывается только представление flights\_v, а таблицу tickets\_directions

* предложение FROM включать не нужно, хотя она и участвует в выполнении соедине-ния таблиц. Конечно, в предложении SET присваивать новые значения можно только атрибутам таблицы tickets\_directions, поскольку именно она приведена в пред-ложении 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 таб-лиц tickets\_directions и flights\_v, можно включить в команду UPDATE пред-ложение RETURNING \*.

223

*Глава 7. Изменение данных*

**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 орга-низовать запись выполненных операций в журнальную таблицу. Эта команда анало-гична команде, уже рассмотренной в предыдущем разделе. В ней также «полезная» работа выполняется в общем табличном выражении, а запись в журнальную табли-цу — 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 за-просу, 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

224

*7.3. Удаление строк из таблиц*

-[ 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 информа-цию 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 пред-ложении WHERE записать условия соединения таблиц. Поскольку первый способ яв-ляется 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 даль-ности полета. Те модели, ранг которых окажется равным 1, будут иметь наименьшую дальность полета.

* предложении USING сформируем соединение таблицы aircrafts\_tmp с времен-ной таблицей min\_ranges, а затем в предложении WHERE зададим условия для отбо-ра 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 \*;**

225

*Глава 7. Изменение данных*

Мы включили в команду DELETE предложение RETURNING \* для того, чтобы пока-зать, как выглядят комбинированные строки, сформированные с помощью предло-жения USING. Конечно, удаляются не они, 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)

* 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 и более сложные выражения, сформированные на основе этих столбцов. В тексте главы мы копировали содержимое таблицы «Самолеты» в таблицу aircrafts\_tmp, используя в предложении RETURNING 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?

226

*Контрольные вопросы и задания*

1. Если бы мы для копирования данных 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 мож-но использовать выведенный результат?

1. В тексте главы в предложениях ON CONFLICT команды INSERT мы использова-ли 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 преду-смотрите возможный конфликт добавляемой строки со строкой, уже имеющей-ся в таблице. Сделайте два варианта предложения ON CONFLICT: первый — с ис-пользованием перечисления имен столбцов для проверки наличия дублирова-ния, 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.

1. В предложении DO UPDATE команды INSERT может использоваться и условие WHERE. Самостоятельно ознакомьтесь с этой возможностью с помощью доку-ментации и напишите такую команду INSERT.
2. Команда COPY по умолчанию ожидает получения вводимых данных в формате text, когда значения данных разделяются символами табуляции. Однако мож-но 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 '\.'.

227

*Глава 7. Изменение данных*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **IL9,** | **Ilyushin IL96, 9800** | |  |  |  |
| **I93, Ilyushin IL96-300,** | | | **9800** |  |  |
| **\.** |  |  |  |  |  |
| COPY | 2 |  |  |  |  |
| **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 зна-чения в столбце model оказались смещены вправо?

1. Команда COPY позволяет получить входные данные из файла и поместить их в таблицу. Этот файл должен быть доступен тому пользователю операцион-ной 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 проверяются все ограничения це-лостности, наложенные на таблицу, то дублирующие строки добавлены, конеч-но же, не будут. А как вы думаете, строки, содержащиеся в этом же файле, но отсутствующие в таблице, будут добавлены или нет?

228

*Контрольные вопросы и задания*

Проверьте свою гипотезу, выполнив вставку 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 биле-тов, проданных по всем направлениям на текущую дату. Однако тот запрос был рассчитан на одновременное добавление только одной записи в таблицу «Пере-леты» (ticket\_flights\_tmp). Ниже мы предложим более универсальный за-прос, который предусматривает возможность единовременного ввода несколь-ких 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 пе-риода 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 плани-рует совершить перелеты по маршруту: Красноярск — Москва, Москва — Сочи, Сочи — Москва, Москва — Сочи, Сочи — Красноярск. Выполнив вспомогатель-ный запрос, выберем следующие идентификаторы рейсов (в этом же порядке): 13829, 4728, 30523, 7757, 30829.

229

*Глава 7. Изменение данных*

**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

* этой версии запроса предусмотрен единовременный ввод нескольких строк в таблицу ticket\_flights\_tmp, причем перелеты могут выполняться на раз-личных рейсах. Поэтому необходимо преобразовать список идентификаторов этих рейсов в множество пар «город отправления — город прибытия», посколь-ку именно для таких пар и ведется подсчет числа забронированных перелетов. Эта задача решается в предложении WHERE, где вложенный подзапрос форми-рует список идентификаторов рейсов, а внешний подзапрос преобразует этот список в множество пар «город отправления — город прибытия». Затем с помо-щью 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 подза-прос: он выполняется для каждой строки, отобранной в предложении WHERE.

* нем используется соединение временной таблицы sell\_tickets с представ-лением flights\_v. Это нужно для того, чтобы подсчитать все перелеты, соот-

230

*Контрольные вопросы и задания*

ветствующие паре атрибутов «город отправления — город прибытия», взятых из текущей обновляемой строки таблицы tickets\_directions. Этот подза-прос позволяет учесть такой факт: рейсы могут иметь различные идентифика-торы flight\_id, но при этом соответствовать одному и тому же направлению, а в таблице tickets\_directions учитываются именно направления.

* случае попытки повторного бронирования одного и того же перелета для дан-ного 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 отра-жено 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)

231

*Глава 7. Изменение данных*

**Задание.** Модифицируйте запрос и таблицуtickets\_directionsтак, чтобыучет числа забронированных перелетов по различным маршрутам выполнялся для каждого класса обслуживания: Economy, Business и Comfort.

9.\* Предположим, что руководство нашей авиакомпании решило отказаться от ис-пользования самолетов компаний Boeing и Airbus, имеющих наименьшее ко-личество 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 выпол-няется *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 удо-влетворяют следующим требованиям: модель — Boeing или Airbus, а число мест

* 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 \*;**

232

*Контрольные вопросы и задания*

Шаг 1 выполняется в подзапросе в предложении WITH. Шаг 2 — в главном запро-се 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 про-бельного символа (используем функции left и strpos). Мы включили предло-жение RETURNING \*, 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 (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

233

*Глава 7. Изменение данных*

Обратите внимание, что в результате были выведены комбинированные стро-ки, полученные при соединении таблицы aircrafts\_tmp с временной табли-цей aicrafts\_seats, указанной в предложении USING. Но удалены были, ко-нечно, строки из таблицы aircrafts\_tmp.

**Задание.** Предложите другой вариант решения этой задачи. Например, можнопоступить 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 удаляе-мых моделей самолетов.

Еще один вариант решения задачи связан с использованием представлений, ко-торые мы рассматривали в главе 5. Можно создать представление на основе таблиц «Самолеты» (aircrafts) и «Места» (seats) и перенести конструкцию

* функциями left и strpos в представление. В нем будут вычисляемые столб-цы: 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 функ-ция min.

234

*Контрольные вопросы и задания*

10.\* В реальной работе иногда возникают ситуации, когда требуется быстро за-полнить таблицу тестовыми данными. В таком случае удобно воспользоваться командой INSERT с подзапросом. Конечно, число атрибутов и их типы данных

* 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 сна-чала 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 фор-мироваться результирующие значения для вставки в таблицу «Места».

* рамках реляционной модели наиболее естественным будет представление ис-ходных данных в виде таблиц. Для формирования каждой строки таблицы «Ме-ста» нужно задать код модели самолета, класс обслуживания и номер места, который состоит из двух компонентов: номера ряда и буквенного идентифи-катора позиции в ряду.

Поскольку размеры и компоновки салонов различаются, необходимо для каж-дой 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 последне-го 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 бук-ва F. В салоне бизнес-класса число мест в ряду меньше, чем в салоне экономи-ческого класса, но для упрощения задачи примем эти числа одинаковыми.

* + результате получим первую исходную таблицу с атрибутами:

– код модели самолета;

– номер последнего ряда кресел в салоне бизнес-класса;

235

*Глава 7. Изменение данных*

– 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.

Так же поступим и с буквенными обозначениями кресел в ряду. В этой таблице будет один атрибут — латинская буква, обозначающая позицию кресла.

* принципе можно было бы создать все четыре таблицы с помощью команды CREATE TABLE и ввести в них исходные данные, а затем использовать эти таб-лицы в команде SELECT. Но команда SELECT позволяет использовать в предло-жении FROM виртуальные таблицы, которые можно создавать с помощью пред-ложения 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 получа-ет имя таблицы, и к ней прилагается список атрибутов. Это выглядит, например, следующим образом:

**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 список иденти-фикаторов — имена ее атрибутов (aircraft\_code, max\_seat\_row\_business, max\_seat\_row\_economy, max\_letter). Эти атрибуты можно использовать во всех частях команды SELECT, как если бы это были атрибуты обычной таблицы.

236

*Контрольные вопросы и задания*

Остальные виртуальные таблицы создаются аналогичным способом.

Для соединения таблиц используется ключевое слово CROSS JOIN, хотя в дан-ном 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.

* этом условии используется условный оператор CASE. Он позволяет нам поста-вить 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**

* этом выражении используется приведение типов: seat\_row::integer. Эта операция 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. По-этому 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 выра-жений, которые будут выведены в качестве итоговых данных. Для формирова-ния 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 «Ме-ста» сразу необходимое число строк, выглядит так:

237

*Глава 7. Изменение данных*

**INSERT INTO seats ( aircraft\_code, seat\_no, fare\_conditions ) SELECT aircraft\_code, seat\_row || letter, fare\_condition**

**FROM**

**-- компоновки салонов**

* **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 )**

**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;**

238

*Контрольные вопросы и задания*

**Задание.** Модифицируйте команду с учетом того, что в салоне бизнес-классачисло мест в ряду должно быть меньше, чем в салоне экономического класса (в приведенном решении мы для упрощения задачи принимали эти числа оди-наковыми).

Попробуйте упростить подзапрос, отвечающий за формирование списка номе-ров рядов кресел:

**( VALUES ( '1' ), ( '2' ), ( '3' ), ( '4' ), ( '5' ), ...**

Воспользуйтесь функцией generate\_series, описанной в разделе документации 9.24 «Функции, возвращающие множества».

239

**Глава 8**

**Индексы**

Индексы позволяют повысить производительность базы данных. PostgreSQL поддерживает различ-ные типы индексов. Мы ограничимся рассмотрением только индексов на основе B-дерева. Индекс — специальная структура данных, которая связана с таблицей и создается на основе данных, содержа-щихся в ней. Основная цель создания индексов — повышение производительности функционирова-ния 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 уско-рения этого поиска и создается индекс. В принципе он организован таким образом: на основе данных, содержащихся в конкретной строке таблицы, формируется зна-чение элемента (записи) индекса, соответствующего этой строке. Для поддержания соответствия между элементом индекса и строкой таблицы в каждый элемент поме-щается 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 соот-ветствующей строке таблицы по прямой ссылке. Записи индекса могут формировать-ся на основе значений одного или нескольких полей соответствующих строк таблицы. Значения этих полей могут комбинироваться и преобразовываться различными спо-собами. 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, со-зданный 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 целесооб-разно, то сначала выполняется поиск необходимых значений в индексе, а затем, если такие значения в нем найдены, производится обращение к таблице с использовани-ем указателей, которые хранятся в записях индекса. Таким образом, полный перебор

241

*Глава 8. Индексы*

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 со-здание 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 предусматрива-лось 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 одинаковы-ми 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 ав-томатически создает индекс, который используется для обеспечения уникальности значений.

Для того чтобы увидеть индексы, созданные для данной таблицы, нужно воспользо-ваться 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.

242

*8.1. Общая информация*

* описании также присутствует список столбцов, по которым создан индекс, и тип индекса — в данном случае это btree, т. е. B-дерево. PostgreSQL может создавать ин-дексы различных типов, но по умолчанию используется так называемое B-дерево. Такой индекс подходит для большинства типовых задач. В этой главе мы будем рас-сматривать только индексы на основе B-дерева.

Наличие индекса может ускорить выборку строк из таблицы, если он создан по столб-цам, 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, ...);**

* этой команде имя индекса можно не указывать. В качестве примера давайте созда-дим индекс для таблицы «Аэропорты» 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**

243

*Глава 8. Индексы*

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 сна-чала 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.

244

*8.1. Общая информация*

**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 значи-тельно меньше.

Просмотреть список всех индексов в текущей базе данных можно командой

**\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. Ин-декс, созданный по столбцу, участвующему в соединении двух таблиц, может позво-лить ускорить процесс выборки записей из таблиц. При выборке записей в отсорти-рованном порядке индекс также может помочь, если сортировка выполняется по тем столбцам, по которым индекс создан.

245

*Глава 8. Индексы*

**8.2. Индексы по нескольким столбцам**

Индексы могут создаваться не только по одному столбцу, но и сразу по несколь-ким. 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 избе-жать 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

* 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 низка), тогда наличие индекса может не дать ожидаемого эффекта. Индексы более полезны, когда из табли-цы выбирается лишь небольшая доля строк, т. е. при *высокой селективности* выборки.
* случае использования предложения ORDER BY в комбинации с LIMIT *n* явная сорти-ровка (при отсутствии индекса) потребует обработки всех строк таблицы ради того, чтобы определить первые *n* строк. Но если есть индекс по тем же столбцам, по кото-рым производится сортировка ORDER BY, то эти первые *n* строк могут быть извлечены непосредственно, без сканирования остальных строк вообще.

Если для таблицы «Билеты» еще не создан индекс по столбцу book\_ref, 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, вероят-но, на два порядка.

246

*8.3. Уникальные индексы*

При создании индексов может использоваться не только возрастающий порядок зна-чений в индексируемом столбце, но и убывающий. По умолчанию порядок возраста-ющий, при этом значения NULL, которые также могут присутствовать в индексируе-мых столбцах, идут последними. При создании индекса можно модифицировать по-ведение 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 ат-рибутов 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»:

**CREATE UNIQUE INDEX aircrafts\_unique\_model\_key**

**ON aircrafts ( model );**

* этом случае мы уже не сможем ввести в таблицу aircrafts строки, имеющие оди-наковые наименования моделей самолетов. Конечно, мы могли при создании табли-цы 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 совпа-дающими считаются лишь те комбинации значений атрибутов в двух строках, в ко-торых совпадают значения всех соответствующих атрибутов.

247

*Глава 8. Индексы*

**8.4. Индексы на основе выражений**

* команде создания индекса можно использовать не только имена столбцов, но так-же функции и скалярные выражения, построенные на основе столбцов таблицы. На-пример, 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 стро-ится 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 поддер-жания при вставке и обновлении строк в таблице. Зато при выполнении выборок, построенных на основе сложных выражений, работа происходит с меньшими на-кладными расходами, поскольку в индексе хранятся уже вычисленные значения этих выражений, пусть даже самых сложных. Поэтому такие индексы целесообразно ис-пользовать 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.

248

*8.5. Частичные индексы*

Это достигается с помощью использования условного выражения, называемого *пре-дикатом индекса*. Предикат вводится с помощью предложенияWHERE.

* качестве иллюстрации создадим частичный индекс для таблицы «Бронирования». 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,

* 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 форми-ровании 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 поря-док меньшее, чем без использования частичного индекса.

* разделе документации 11.8 «Частичные индексы» сказано, что для того, чтобы СУБД использовала частичный индекс, необходимо, чтобы условие, записанное в за-просе в предложении WHERE, соответствовало предикату индекса. Это означает, что

249

*Глава 8. Индексы*

либо условие должно быть точно таким же, как использованное в предикате частич-ного индекса при его создании, либо условие запроса должно математически сво-диться 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. за-дание 9). Однако размер частичного индекса будет меньше, чем размер обычного. Для получения заметного полезного эффекта от их применения необходимы 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 ат-рибута column1 равно ABC, а значение атрибута column2 — NULL. Мы решили добавить в таблицу еще одну строку с такими же значениями ключевых атри-бутов, т. е. column1 — ABC, а column2 — NULL.

Как вы думаете, будет ли операция вставки новой строки успешной или завер-шится с ошибкой? Объясните ваше решение.

1. В тексте главы шла речь о выполнении одной и той же выборки из таблицы «Би-леты» (tickets) при наличии индекса по столбцу passenger\_name и при его отсутствии. Вы видели, что наличие индекса ускоряет выполнение запроса по-чти на порядок.

Если секундомер в утилите psql выключен, то включите его с помощью команды

**\timing on**

250

*Контрольные вопросы и задания*

Проведите следующий эксперимент: выполните этот запрос несколько раз под-ряд 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 сокращает-ся, причем, когда создан индекс, оно сокращается на порядок. Как вы думаете, почему?

1. Известно, что индекс значительно ускоряет работу, если при выполнении за-проса из таблицы отбирается лишь небольшая часть строк. Если же эта доля велика, скажем, половина строк или более, то большого положительного эффек-та 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 имеется столбец «Класс обслуживания» (fare\_conditions), который отличается от остальных 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 изме-няться, и подсчитайте среднее время. Обратите внимание на число строк, ко-торые возвращает функция count для каждого значения атрибута. При этом среднее время выполнения запросов для трех различных значений атрибута fare\_conditions будет различаться незначительно, поскольку в каждом слу-чае СУБД просматривает все строки таблицы.

**SELECT count( \* ) FROM ticket\_flights**

**WHERE fare\_conditions = 'Comfort';**

**SELECT count( \* ) FROM ticket\_flights**

**WHERE fare\_conditions = 'Business';**

251

*Глава 8. Индексы*

**SELECT count( \* )**

**FROM ticket\_flights**

**WHERE fare\_conditions = 'Economy';**

Создайте индекс по столбцу fare\_conditions. Конечно, в реальной ситуации такой индекс вряд ли целесообразен, но нам он нужен для экспериментов.

Проделайте те же эксперименты с таблицей ticket\_flights. Будет ли разли-чаться среднее время выполнения запросов для различных значений атрибута fare\_conditions? Почему это имеет место?

В завершение этого упражнения отметим, что в случае ошибки планировщи-ка при использовании индекса возможно не только отсутствие положительного эффекта, но и значительный отрицательный эффект.

1. Для одной из таблиц создайте индекс по двум столбцам, причем по одному из них укажите убывающий порядок значений столбца, а по другому — возрастаю-щий. Значения NULL у первого столбца должны располагаться в начале, а у вто-рого — в конце. Посмотрите полученный индекс с помощью команд psql

**\d имя\_таблицы \di+ имя\_индекса**

Обратите внимание, что первая команда выведет не только имя индекса, но так-же 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 мне-нию, не будет. Проверьте ваши гипотезы, выполнив запросы. Объясните полу-ченные результаты.

1. В сложных базах данных целесообразно использование комбинаций индек-сов. Иногда бывают более полезны комбинированные индексы по нескольким столбцам, чем отдельные индексы по единичным столбцам. В реальных ситу-ациях часто приходится делать выбор, т. е. находить компромисс, между, на-пример, 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 возмож-ных решений. Воспользуйтесь документацией на PostgreSQL.

252

*Контрольные вопросы и задания*

1. Предложите какую-нибудь таблицу в базе данных «Авиаперевозки» и смодели-руйте 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 неко-торых ситуациях бывает целесообразно создавать индекс по столбцам внешне-го 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 отноше-нии которых было бы целесообразно поступить так, как говорится в докумен-тации.

8.\* В тексте главы был показан пример использования частичного индекса для таб-лицы «Бронирования». 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 ин-декса 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 ис-пользовании частичного индекса. Очень вероятно, что различия времени вы-полнения запроса будут незначительными.

Самостоятельно ознакомьтесь с разделом документации 11.8 «Частичные ин-дексы» и попробуйте смоделировать ситуацию в предметной области «Авиапе-ревозки», когда частичный индекс дал бы больший эффект, чем обычный ин-декс.

253

*Глава 8. Индексы*

1. Когда выполняются запросы с поиском по шаблону LIKE или регулярными вы-ражениями POSIX, тогда для того, чтобы использовался индекс, нужно преду-смотреть 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 операторов, 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 за-просов. Поэтому, возможно, потребуется создать еще и индекс с классом опера-торов по умолчанию. Самостоятельно изучите этот вопрос с помощью раздела документации 11.9 «Семейства и классы операторов».

254

**Глава 9**

**Транзакции**

Детальное понимание механизмов выполнения транзакций придет с опытом. В этой главе мы дадим самое первое представление об этом важном и мощном инструменте, которым обладают все серьез-ные СУБД, включая PostgreSQL.

**9.1. Общая информация**

Транзакция — это совокупность операций над базой данных, которые вместе образу-ют логически целостную процедуру, и могут быть либо выполнены все вместе, либо не будет выполнена ни одна из них. В простейшем случае транзакция состоит из од-ной операции.

Транзакции являются одним из средств обеспечения согласованности (непротиворе-чивости) базы данных, наряду с ограничениями целостности (constraints), наклады-ваемыми на таблицы. Транзакция переводит базу данных из одного согласованного состояния в другое согласованное состояние.

* качестве примера транзакции в базе данных «Авиаперевозки» можно привести процедуру бронирования билета. Она будет включать операции INSERT, выполня-емые над таблицами «Бронирования» (bookings), «Билеты» (tickets) и «Переле-ты» (ticket\_flights). В результате выполнения этой транзакции должно обеспечи-ваться 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 дан-ной транзакции будут выполнены частично, тогда может оказаться, например, что общая сумма бронирования будет не равна сумме стоимостей перелетов, включен-ных в это бронирование. Очевидно, что это несогласованное состояние базы данных.

Транзакция может иметь два исхода: первый — изменения данных, произведенные

* ходе ее выполнения, успешно зафиксированы в базе данных, а второй исход таков — транзакция отменяется, и отменяются все изменения, выполненные в ее рамках. От-мена транзакции называется откатом (rollback).

255

*Глава 9. Транзакции*

Сложные информационные системы, как правило, предполагают одновременную ра-боту многих пользователей с базой данных, поэтому современные СУБД предлагают специальные механизмы для организации параллельного, т. е. одновременного, вы-полнения 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 вре-мени. При этом параллельно исполняемые транзакции, даже вносящие изменения

* базу данных, не нарушают согласованности данных этого снимка. Такой результат
* PostgreSQL достигается за счет того, что когда параллельные транзакции изменяют одни и те же строки таблиц, тогда создаются отдельные *версии* этих строк, доступ-ные 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 за-фиксирована 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 со-гласованного состояния в другое согласованное состояние.
3. Изолированность (isolation). Во время выполнения транзакции другие транзак-ции 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 впослед-ствии 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 возможны следующие феномены:

1. Потерянное обновление (lost update). Когда разные транзакции одновременно изменяют одни и те же данные, то после фиксации изменений может оказаться,

256

*9.1. Общая информация*

что одна транзакция перезаписала данные, обновленные и зафиксированные другой транзакцией.

1. «Грязное» чтение (dirty read). Транзакция читает данные, измененные парал-лельной транзакцией, которая еще не завершилась. Если эта параллельная транзакция в итоге будет отменена, тогда окажется, что первая транзакция про-читала данные, которых нет в системе.
2. 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.
3. Фантомное чтение (phantom read). Транзакция повторно выбирает множество строк в соответствии с одним и тем же критерием. В интервале времени меж-ду выполнением этих выборок другая транзакция добавляет новые строки и успешно фиксирует изменения. В результате при выполнении повторной вы-борки в первой транзакции может быть получено другое множество строк.
4. 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 рас-смотрены подробно и проиллюстрированы примерами.

Поясним кратко, в чем состоит смысл концепции сериализации. Для двух транзак-ций, скажем, A и B, возможны только два варианта упорядочения при их последова-тельном выполнении: сначала A, затем B или сначала B, затем A. Причем результаты реализации двух вариантов могут в общем случае не совпадать. Например, при вы-полнении 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 результа-ты, зависящие от порядка их выполнения.

Сериализация двух транзакций при их *параллельном* выполнении означает, что полу-ченный результат будет соответствовать *одному* из двух возможных вариантов упоря-дочения транзакций при их последовательном выполнении. При этом нельзя сказать точно, какой из вариантов будет реализован.

257

*Глава 9. Транзакции*

Если распространить эти рассуждения на случай, когда параллельно выполняется бо-лее 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 сериа-лизации не предписывает выбора какого-то определенного варианта. Речь идет лишь об одном из них.

* 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 по-нятие 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) дан-ных. Однако в PostgreSQL требования, предъявляемые к этому уровню, более строгие, чем в стандарте: чтение «грязных» данных на этом уровне не допуска-ется.
2. Read Committed. Не допускается чтение «грязных» (незафиксированных) дан-ных. Таким образом, в PostgreSQL уровень Read Uncommitted совпадает с уров-нем 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) дан-ных и неповторяющееся чтение. В PostgreSQL на этом уровне не допускается также фантомное чтение. Таким образом, реализация этого уровня является бо-лее 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 внутрен-них механизмов. Его достаточно указать в команде при старте транзакции. Одна-ко программист может дополнительно использовать некоторые операторы и прие-мы программирования, например, устанавливать блокировки на уровне отдельных строк или всей таблицы. Это будет показано в конце главы.

258

*9.2. Уровень изоляции Read Uncommitted*

По умолчанию PostgreSQL использует уровень изоляции Read Committed.

**SHOW default\_transaction\_isolation;**

default\_transaction\_isolation

-------------------------------

read committed

(1 строка)

**9.2. Уровень изоляции Read Uncommitted**

Давайте начнем рассмотрение с уровня изоляции Read Uncommitted. Проверим, ви-дит ли транзакция те изменения данных, которые были произведены в другой тран-закции, 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 удаля-лись строки из таблицы «Места» (seats), связанные по внешнему ключу со строками из таблицы aircrafts.

**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 экспе-рименты.

На первом терминале выполним следующие команды:

**BEGIN;**

BEGIN

**SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;**

SET

259

*Глава 9. Транзакции*

**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;**

SET

|  |  |  |  |
| --- | --- | --- | --- |
| **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

260

*9.3. Уровень изоляции Read Committed*

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 восполь-зуемся командой ROLLBACK для отмены транзакции, т. е. для ее отката.

На первом терминале:

**ROLLBACK;**

ROLLBACK

На втором терминале сделаем так же:

**ROLLBACK;**

ROLLBACK

**9.3. Уровень изоляции Read Committed**

Теперь обратимся к уровню изоляции Read Committed. Именно этот уровень уста-новлен 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 воз-можно 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)

261

*Глава 9. Транзакции*

**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 изме-нено, хотя пока и не зафиксировано. Но транзакция видит изменения, выполненные в ней самой.

Обратите внимание, что вместо использования команды SET TRANSACTION мы про-сто 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';**

* вот мы видим, что команда UPDATE во второй транзакции не завершилась, а пере-шла в состояние ожидания. Это ожидание продлится до тех пор, пока не завершится первая транзакция. Дело в том, что команда UPDATE в первой транзакции заблоки-ровала 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.

262

*9.3. Уровень изоляции Read Committed*

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 транзак-ции, получив возможность заблокировать строку после завершения первой транзак-ции и снятия ею блокировки с этой строки, *перечитывает* строку таблицы и потому обновляет строку, уже обновленную в только что зафиксированной транзакции. Та-ким образом, эффекта потерянных обновлений не возникает.

Завершим транзакцию на втором терминале, но вместо команды COMMIT воспользу-емся 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 про-стой эксперимент также на двух терминалах. На первом терминале:

**BEGIN;**

BEGIN

263

*Глава 9. Транзакции*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 | | | | 3300 |
| (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 | | | 3300 |

(6 строк)

Сразу завершим вторую транзакцию:

**END;**

COMMIT

264

*9.4. Уровень изоляции Repeatable Read*

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 | | | 3300 |

(6 строк)

Видим, что теперь получен другой результат, т. к. вторая транзакция завершилась

* момент времени между двумя запросами. Таким образом, налицо эффект неповто-ряющегося чтения данных, который является допустимым на уровне изоляции Read Committed.

Завершим и первую транзакцию:

**END;**

COMMIT

**9.4. Уровень изоляции Repeatable Read**

Третий уровень изоляции — Repeatable Read. Само его название говорит о том, что он не допускает феномен 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, ис-пользующая этот уровень изоляции, создает снимок данных не перед выполнением каждого запроса, а только однократно, перед выполнением *первого запроса* транзак-ции. Поэтому транзакции с этим уровнем изоляции не могут изменять строки, ко-торые были изменены другими завершившимися транзакциями уже после создания снимка. Вследствие этого PostgreSQL не позволит зафиксировать транзакцию, кото-рая попытается изменить уже измененную строку.

265

*Глава 9. Транзакции*

Важно помнить, что повторный запуск может потребоваться только для транзакций, которые вносят изменения в данные. Для транзакций, которые только читают дан-ные, 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 | | | | 3300 |
| 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 и одну строку обновим:

**INSERT INTO aircrafts\_tmp**

**VALUES ( 'IL9', 'Ilyushin IL96', 9800 );**

INSERT 0 1

**UPDATE aircrafts\_tmp**

**SET range = range + 100**

**WHERE aircraft\_code = '320';**

UPDATE 1

266

*9.4. Уровень изоляции Repeatable Read*

**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 | | | 3300 |
| 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 | | | 3300 |
| CN1 | | | Cessna | 208 Caravan | | | 2100 |
| CR2 | | | Bombardier CRJ-200 | | | | 1900 |
| IL9 | | | Ilyushin IL96 | | | | 9800 |
| 320 | | | Airbus | A320-200 | | | 5800 |

(7 строк)

267

*Глава 9. Транзакции*

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 терми-нале 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,

268

*9.5. Уровень изоляции Serializable*

что строка уже изменена. На уровне изоляции Repeatable Read снимок данных созда-ется на момент начала выполнения первого запроса транзакции и в течение тран-закции 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 об-новление, что недопустимо. В результате генерируется ошибка, и вторая транзакция откатывается. Мы вводим команду END на втором терминале, но PostgreSQL выпол-няет 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

---------------+-----------------+-------

1. | Airbus A320-200 | 5900

(1 строка)

**9.5. Уровень изоляции Serializable**

Самый высший уровень изоляции транзакций — Serializable. Транзакции могут ра-ботать параллельно точно так же, как если бы они выполнялись последовательно одна за другой. Однако, как и при использовании уровня Repeatable Read, прило-жение должно быть готово к тому, что придется перезапускать транзакцию, кото-рая 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 возмож-ных вариантов* их упорядочения, если бы они выполнялись последовательно, одна задругой.

Для проведения эксперимента создадим специальную таблицу, в которой будет всего два столбца: один — числовой, а второй — текстовый. Назовем эту таблицу modes.

269

*Глава 9. Транзакции*

**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

* команде обновления строки будем использовать предложение RETURNING. По-скольку значение поля num не изменяется, то будет видно, какая строка была обнов-лена. 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.

270

*9.5. Уровень изоляции Serializable*

**BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;**

BEGIN

**UPDATE modes**

**SET mode = 'LOW'**

**WHERE mode = 'HIGH'**

**RETURNING \*;**

num | mode

-----+------

2|LOW

(1 строка)

UPDATE 1

Изменение, произведенное в первой транзакции, вторая транзакция не видит, по-скольку на уровне изоляции Serializable каждая транзакция работает с тем снимком базы данных, который был сделан непосредственно перед выполнением ее перво-го оператора. Поэтому обновляется только одна строка, та, в которой значение поля mode было равно HIGH изначально.

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)

271

*Глава 9. Транзакции*

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 уда-лось. 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 последо-вательно. Следовательно, с точки зрения концепции сериализации эти транзакции невозможно сериализовать.

Покажем это, выполнив транзакции последовательно.

272

*9.5. Уровень изоляции Serializable*

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

273

*Глава 9. Транзакции*

Проверим, что получилось:

**SELECT \* FROM modes;**

num | mode

-----+------

2|LOW

1|LOW

(2 строки)

Во втором варианте упорядочения поменяем транзакции местами. Конечно, предва-рительно 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 \*;**

274

*9.6. Пример использования транзакций*

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. Одна-ко если бы при параллельном выполнении транзакций была зафиксирована и вторая из них, то полученный результат не соответствовал бы *ни одному* из продемонстриро-ванных возможных результатов последовательного выполнения транзакций. Таким образом, выполнить сериализацию этих транзакций невозможно. Обратите внима-ние, что вторая команда UPDATE в обоих случаях обновляет не одну строку, а две.

**9.6. Пример использования транзакций**

Продемонстрируем использование транзакций на примере базы данных «Авиапе-ревозки». Для этого создадим новое бронирование и оформим два билета с двумя перелетами в каждом. Выберем в качестве уровня изоляции Read Committed.

**BEGIN;**

BEGIN;

275

*Глава 9. Транзакции*

Сначала добавим запись в таблицу «Бронирования», причем назначим значение по-ля total\_amount равным 0. После завершения ввода строк в таблицу «Перелеты» мы обновим это значение: оно станет равным сумме стоимостей всех забронирован-ных перелетов. В качестве даты бронирования возьмем дату, которая была принята

* 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 таб-лицы «Бронирования». Конечно, если такая транзакция выполняется в рамках при-кладной программы, то возможно, что подсчет общей суммы будет выполняться

* этой программе. Тогда в команде UPDATE уже не потребуется выполнять подзапрос, а будет использоваться заранее вычисленное значение. Но более надежным решени-ем было бы использование триггера для увеличения значения поля total\_amount

276

*9.6. Пример использования транзакций*

при каждом добавлении строки в таблицу ticket\_flights, но в этом учебном по-собии они не рассматриваются.

**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;

* начале главы говорилось о свойствах транзакций. Их удобно прокомментировать на примере этой транзакции, в которой участвуют три таблицы. *Атомарность* го-ворит 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 на-ходилась до начала транзакции. При выборе соответствующего уровня *изоляции* эта транзакция сможет выполняться, не подвергаясь помехам со стороны других парал-лельных 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.

277

*Глава 9. Транзакции*

**9.7. Блокировки**

Кроме поддержки уровней изоляции транзакций, PostgreSQL позволяет также созда-вать явные блокировки данных как на уровне отдельных строк, так и на уровне це-лых таблиц. Блокировки могут быть востребованы при проектировании транзакций

* 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 транзак-ция заблокировала строки с помощью этой команды, тогда параллельные транзакции не смогут заблокировать эти же строки до тех пор, пока первая транзакция не завер-шится, и тем самым блокировка не будет снята.

Проведем эксперимент, как и прежде, с использованием двух терминалов. Мы не бу-дем 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

---------------+-----------------+-------

1. | Airbus A320-200 | 5700
2. | Airbus A321-200 | 5600
3. | Airbus A319-100 | 6700

(3 строки)

На втором терминале организуйте аналогичную транзакцию и выполните точно та-кую же команду. Вы увидите, что ее выполнение будет приостановлено.

**SELECT \***

**FROM aircrafts\_tmp**

**WHERE model ~ '^Air'**

**FOR UPDATE;**

278

*9.7. Блокировки*

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 выбор-ка, которая показала уже измененные данные:

aircraft\_code | model | range

---------------+-----------------+-------

1. | Airbus A320-200 | 5800
2. | Airbus A321-200 | 5600
3. | 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 задержа-но. 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»).

279

*Глава 9. Транзакции*

**Контрольные вопросы и задания**

1. По умолчанию каждая SQL-команда, выполняемая в среде psql, образует от-дельную транзакцию с уровнем изоляции Read Committed. Поэтому в тех экс-периментах, 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 рассматрива-лись команды SELECT ... FOR UPDATE и LOCK TABLE. Попробуйте повторить эти эксперименты с учетом описанного поведения PostgreSQL.

1. 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 ситу-ации, которые на первый взгляд кажутся парадоксальными, но на самом деле все происходит в строгом соответствии с этим принципом.

Воспользуемся таблицей «Самолеты» (aircrafts) или ее копией. Предполо-жим, что мы решили удалить из таблицы те модели, дальность полета которых менее 2 000 км. В таблице представлена одна такая модель — Cessna 208 Caravan, имеющая дальность полета 1 200 км. Для выполнения удаления мы организова-ли транзакцию. Однако параллельная транзакция, которая, причем, началась раньше, успела обновить таблицу таким образом, что дальность полета самоле-та Cessna 208 Caravan стала составлять 2 100 км, а вот для самолета Bombardier CRJ-200 она, напротив, уменьшилась до 1 900 км. Таким образом, в результате выполнения операций обновления в таблице по-прежнему присутствует стро-ка, 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

280

*Контрольные вопросы и задания*

**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, установ-ленная командой UPDATE в первой транзакции, снимается только при завер-шении транзакции, а завершение может иметь два исхода: фиксацию измене-ний с помощью команды COMMIT (или END) или отмену изменений с помощью команды ROLLBACK.

281

*Глава 9. Транзакции*

Давайте зафиксируем изменения, выполненные первой транзакцией. На пер-вом 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 стро-ки, удовлетворяющей условию удаления.

Объяснение таково: поскольку вторая транзакция пока еще не видит измене-ний, 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 из-менила 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 воз-можность заблокировать эту строку. При этом команда DELETE данную строку *перечитывает* и вновь вычисляет условиеWHEREприменительно к ней. Однакотеперь 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 по-вторный поиск строк, удовлетворяющих условию WHERE в команде DELETE, не производится.

* результате не удаляется ни одна строка. Таким образом, к сожалению, имеет место нарушение согласованности, которое можно объяснить деталями реали-зации СУБД.

Завершим вторую транзакцию:

**END;**

COMMIT

Вот что получилось в результате:

**SELECT \* FROM aircrafts\_tmp;**

282

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | *Контрольные вопросы и задания* |
| 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: в первой тран-закции 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 ка-честве примера приводится операция UPDATE, в которой значение какого-то атрибута изменяется с применением одного из действий арифметики. Напри-мер:

**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, бу-дут учтены все такие изменения (что и было показано в тексте главы). Очевид-но, что потерянного обновления не происходит.

Предположим, что в одной транзакции будет просто присваиваться новое зна-чение, например, так:

**UPDATE aircrafts\_tmp**

**SET range = 2100**

**WHERE aircraft\_code = 'CR2';**

А в параллельной транзакции будет выполняться аналогичная команда:

**UPDATE aircrafts\_tmp**

**SET range = 2500**

**WHERE aircraft\_code = 'CR2';**

283

*Глава 9. Транзакции*

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 фундамен-тальному 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.

1. На уровне изоляции транзакций Read Committed имеет место такой феномен, как чтение фантомных строк. Такие строки могут появляться в выборке как в ре-зультате добавления новых строк параллельной транзакцией, так и вследствие изменения ею значений атрибутов, участвующих в формировании условия вы-борки. Рассмотрим пример, иллюстрирующий вторую из указанных причин.

На первом терминале организуем транзакцию. Она будет иметь уровень изоля-ции Read Committed:

**BEGIN;**

BEGIN

**SELECT \***

**FROM aircrafts\_tmp WHERE range > 6000;**

aircraft\_code | model | range

---------------+-----------------+-------

1. | Boeing 777-300 | 11100
2. | Boeing 767-300 | 7900
3. | Airbus A319-100 | 6700

(3 строки)

На втором терминале организуем транзакцию и обновим одну из строк табли-цы таким образом, чтобы эта строка стала удовлетворять условию отбора строк, заданному в первой транзакции.

**BEGIN;**

BEGIN

284

*Контрольные вопросы и задания*

**UPDATE aircrafts\_tmp**

**SET range = 6100**

**WHERE aircraft\_code = '320';**

UPDATE 1

Сразу завершим вторую транзакцию, чтобы первая транзакция увидела эти из-менения.

**END;**

COMMIT

На первом терминале повторим ту же самую выборку:

**SELECT \***

**FROM aircrafts\_tmp**

**WHERE range > 6000;**

aircraft\_code | model | range

---------------+-----------------+-------

1. | Boeing 777-300 | 11100
2. | Boeing 767-300 | 7900
3. | Airbus A319-100 | 6700
4. | Airbus A320-200 | 6100

(4 строки)

Транзакция еще не завершилась, но она уже увидела новую строку, обновлен-ную зафиксированной параллельной транзакцией. Теперь эта строка стала соот-ветствовать 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 операцииUPDATEисполь-зуйте операцию INSERT.

1. В тексте главы была рассмотрена команда SELECT ... FOR UPDATE, выполня-ющая блокировку на уровне отдельных строк. Организуйте две параллельные

285

*Глава 9. Транзакции*

транзакции с уровнем изоляции Read Committed и выполните с ними ряд экспе-риментов. В первой транзакции заблокируйте некоторое множество строк, от-бираемых с помощью условия WHERE. А во второй транзакции изменяйте усло-вие 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.

1. 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.
2. В тексте главы для иллюстрации изучаемых концепций мы создавали только две параллельные транзакции. Попробуйте воспроизвести представленные экспе-рименты, 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 исполь-зованием явных блокировок. Обоснуйте ваш ответ.

9.\* В разделе документации 13.2.3 «Уровень изоляции Serializable» сказано, что ес-ли поиск в таблице осуществляется последовательно, без использования индек-са, тогда на всю таблицу накладывается так называемая предикатная блокиров-ка. 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.

286

*Контрольные вопросы и задания*

Добавим в нее такое число строк, которое сделает очень вероятным использо-вание индекса при выполнении операций обновления строк и, соответственно, отсутствие предикатной блокировки всей таблицы. О том, как узнать, исполь-зуется 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

287

*Глава 9. Транзакции*

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 зафиксиро-вать их обе. Как вы думаете, почему это удалось? Обосновывая ваш ответ, при-мите во внимание тот результат, который был бы получен при последователь-ном выполнении транзакций.

288

*Контрольные вопросы и задания*

10.\* В тексте главы был рассмотрен пример транзакции над таблицами базы дан-ных «Авиаперевозки». Давайте теперь создадим две параллельные транзакции

* выполним их с уровнем изоляции Serializable. Отправим также двоих пасса-жиров теми же самыми рейсами, что и ранее, но операции распределим между двумя транзакциями. Отличие заключается в том, что в начале транзакции бу-дут выполняться выборки из таблицы ticket\_flights. Для упрощения ситу-ации 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

289

*Глава 9. Транзакции*

**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

290

*Контрольные вопросы и задания*

**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 сериа-лизовать. Что можно сделать, чтобы удалось зафиксировать обе транзакции? Одно из возможных решений — понизить уровень изоляции. Другим ре-шением 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 командеSELECTтакое:... WHEREflight\_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

* повторите эксперименты, выполнив транзакции параллельно с уровнем изо-ляции 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 последо-вательном выполнении.

291

**Глава 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 таб-лиц и извлечения только тех строк, которые соответствуют критерию отбора. Су-ществуют различные методы доступа: последовательный просмотр (sequential scan), при котором индекс не используется, и группа методов, основанных на использова-нии индекса. К ней относятся: просмотр по индексу (index scan), просмотр исклю-чительно 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 объ-екты считываются в память, в которой они представлены разбитыми на отдельные фрагменты, называемые страницами. Эти страницы имеют специальную структуру. Размер страниц по умолчанию составляет 8 кбайт.

При выполнении **последовательного просмотра** (sequential scan) обращения к ин-дексам не происходит, а строки извлекаются из табличных страниц в соответствии

* критерием отбора. В том случае, когда в запросе нет предложения WHERE, тогда из-влекаются все строки таблицы. Данный метод применяется, когда требуется выбрать все строки таблицы или значительную их часть, т. е. когда так называемая *селектив-ность* выборки низка. В таком случае обращение к индексу не ускорит процесс про-смотра, а возможно даже и замедлит.

**Просмотр на основе индекса** (index scan) предполагает обращение к индексу, со-зданному для данной таблицы. Поскольку в индексе для каждого ключевого значе-ния содержатся уникальные идентификаторы строк в таблицах, то после отыскания в

293

*Глава 10. Повышение производительности*

индексе нужного ключа производится обращение к соответствующей странице таб-лицы и извлечение искомой строки по ее идентификатору. При этом нужно учиты-вать, 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 на-звания 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 тран-закциям — нельзя быть уверенным, что данные, полученные из индекса, видны те-кущей транзакции.

Поэтому сначала выполняется обращение к карте видимости (visibility map), которая существует для каждой таблицы. В ней одним битом отмечены страницы, на кото-рых содержатся только те версии строк, которые видны всем без исключения тран-закциям. Если полученная из индекса версия строки находится на такой странице, значит, эта строка видна текущей транзакции и обращаться к самой таблице не тре-буется. Поскольку размер карты видимости очень мал, то в результате сокращается объем операций ввода/вывода. Если же строка находится на странице, не отмеченной

* карте видимости, тогда происходит обращение и к таблице; в результате никакого выигрыша по быстродействию в сравнении с обычным индексным поиском не до-стигается. Просмотр исключительно на основе индекса особенно эффективен, когда выбираемые данные изменяются редко. Он может применяться, когда в предложе-нии SELECT указаны только имена столбцов, по которым создан индекс.

**Просмотр на основе битовой карты** (bitmap scan) является модификацией про-смотра на основе индекса. Данный метод позволяет оптимизировать индексный по-иск за счет того, что сначала производится поиск в индексе для всех искомых строк

* формирование так называемой битовой карты, в которой указывается, в каких страницах таблицы эти строки содержатся. После того как битовая карта сформиро-вана, 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, опи-санных 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

294

*10.1. Основные понятия*

SQL и является высокоуровневым, логическим, оно не касается вопросов реализа-ции. 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» на-бора и для каждой из них выполняется поиск соответствующих строк во «внутрен-нем» 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 непо-средственной 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 про-цесс обращения к ней. Данный метод работает только при выполнении эквисоедине-ний, поскольку для хеш-таблицы имеет смысл только проверка на равенство прове-ряемого значения одному из ее ключей. Метод эффективен для больших выборок.

**Соединение методом слияния** производится аналогично сортировке слиянием.

* этом случае оба набора строк должны быть предварительно отсортированы по тем столбцам, по которым производится соединение. Затем параллельно читаются стро-ки 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 ме-тод соединения хешированием, работает только при выполнении эквисоединений. Он пригоден для работы с большими наборами строк.

295

*Глава 10. Повышение производительности*

**10.2. Методы просмотра таблиц**

Теперь мы можем перейти к рассмотрению планов выполнения запросов.

Прежде чем приступить к непосредственному выполнению запроса, PostgreSQL фор-мирует *план* его выполнения. Чтобы достичь хорошей производительности, этот план должен учитывать свойства данных. Планированием занимается специальная под-система — планировщик (planner). Просмотреть план выполнения любого запроса можно с помощью команды EXPLAIN. Для детального понимания планов выполне-ния 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 вы-дачу строк таблиц, которые осуществляются с помощью методов доступа, описанных выше. Если конкретный запрос требует выполнения операций агрегирования, соеди-нения таблиц, сортировки, то над узлами выборки строк будут располагаться допол-нительные узлы дерева плана. Например, для соединения наборов строк будут ис-пользоваться способы, которые мы только что рассмотрели. Для каждого узла дерева плана команда EXPLAIN выводит по одной строке, при этом выводятся также оцен-ки стоимости выполнения операций на каждом узле, которые делает планировщик.

* 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.

296

*10.2. Методы просмотра таблиц*

Первое число означает оценку ресурсов, требуемых для того, чтобы приступить к вы-воду данных. В нашем примере эта оценка равна нулю, поскольку никакие дополни-тельные 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 вы-полняется 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,

* каких конкретно единицах производится измерение стоимости: важны соотноше-ния стоимостей. Для каждого запроса планировщик формирует несколько планов. При сравнении различных вариантов плана, как правило, для выполнения выбира-ется тот, который имеет наименьшую общую стоимость выполнения запроса. Однако при работе с курсорами этот принцип можно изменить с помощью специального па-раметра планировщика cursor\_tuple\_fraction (курсоры в учебном пособии не рассматриваются).

Далее в выводе идет общее число строк, которые должны быть извлечены (возвра-щены) на данном узле плана, также при условии выполнения этого узла до полного завершения. В нашем примере число строк равно 9. Это число является оценкой, ко-торую планировщик получает на основе статистики, накапливаемой в специальных системных таблицах.

Последним параметром узла плана идет оценка среднего размера строк, которые вы-водятся на данном узле плана запроса. В нашем примере размер (ширина) строки данных оценивается в 52 байта.

* 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)

297

*Глава 10. Повышение производительности*

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.

* данном случае планировщик неточно оценил число выбираемых строк — фактиче-ски их будет три.

Обратите внимание, что по своей форме вывод команды EXPLAIN также является вы-боркой, поэтому в конце выборки, как обычно, выводится информация о числе строк

* 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)

298

*10.2. Методы просмотра таблиц*

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 последователь-ному сканированию (Seq Scan) таблицы, о чем говорит нам нижний узел плана.

На верхнем узле выполняется сортировка выбранных строк. Поскольку для выполне-ния 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 по-лучения выборки — 1,09.

Когда таблица очень маленькая, то обращение к индексу не даст выигрыша в скоро-сти, а лишь добавит к операциям чтения страниц, в которых хранятся строки таблиц, еще и операции чтения страниц с записями индекса.

Обратимся к таблице «Бронирования» 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 объяс-няется тем, что, хотя индекс уже упорядочен, и дополнительная сортировка не требу-ется, 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 дополни-тельной строке верхнего (и единственного) узла плана.

**EXPLAIN SELECT \***

**FROM bookings**

**WHERE book\_ref > '0000FF' AND book\_ref < '000FFF'**

**ORDER BY book\_ref;**

299

*Глава 10. Повышение производительности*

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 при-мере таблицы «Места».

**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)

* 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 со-ответствии 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;**

300

*10.2. Методы просмотра таблиц*

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)

* этом плане только один узел — Index Only Scan. Здесь также первая оценка стои-мости не нулевая, т. к. отыскание в индексе наименьшего значения требует некото-рого 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)

* верхнем узле плана выполняется агрегирование — Aggregate. А в нижних узлах подготавливаются строки с помощью сканирования на основе формирования бито-вой карты.

Но возникает вопрос: зачем вообще выполняется обращение к страницам таблицы (Bitmap Heap Scan), если никакие значения атрибутов не выбираются, а подсчиты-вается лишь число этих строк? Казалось бы, достаточно использования только ин-декса. Но это нужно для того, чтобы проверить видимость версий строк: ведь разные транзакции могут видеть разные версии строк, поэтому при подсчете их числа нуж-но учитывать, какой транзакции они видны. Обратите еще внимание на тот факт, что собственно стадия агрегирования «стoит»´ 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).

301

*Глава 10. Повышение производительности*

* 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 сформиру-ем запрос, в котором соединяются две таблицы: «Места» и «Самолеты».

**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)

302

*10.3. Методы формирования соединений наборов строк*

Узел Nested Loop, в котором выполняется соединение, имеет два дочерних узла: внешний — Seq Scan и внутренний — Bitmap Heap Scan. Во внешнем узле после-довательно 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)

* е. для текущего значения атрибута aircraft\_code, по которому выполняется со-единение. 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 рей-сы по этим маршрутам. Воспользуемся таблицами «Маршруты» и «Самолеты».

**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) табли-ца aircrafts, и формируется хеш-таблица, ключами которой являются значения ат-рибута 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:

303

*Глава 10. Повышение производительности*

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» и «Перелеты». Он выбирает для каждого билета все перелеты, вклю-ченные в него. Конечно, это очень упрощенный запрос, в реальной ситуации он не представлял бы большой практической пользы, но в целях упрощения плана и повы-шения 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 внима-ние на оценки стоимости выполнения всех трех операций: двух сканирований таблиц

* результирующего соединения слиянием. Мы видим, что первая оценка в узле Merge Join равна 1,51, что значительно меньше вторых оценок, вычисленных планиров-щиком для двух нижних узлов, а именно: 17230,42 и 67058,74.

304

*10.4. Управление планировщиком*

Напомним, что первая оценка говорит, сколько ресурсов будет затрачено (сколько времени, в условных единицах, пройдет) до начала вывода первых результатов вы-полнения операции на данном уровне дерева плана. Вторая оценка показывает об-щее 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 из-менить на время текущего сеанса работы с помощью команды SET. Конечно, изме-нять параметры в производственной базе данных следует только в том случае, когда вы *обоснованно* считаете, что планировщик ошибается. Однако для того чтобы на-учиться видеть ошибки планировщика, нужен большой опыт. Поэтому следует рас-сматривать приведенные далее команды управления планировщиком лишь с пози-ции 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 осно-ве хеширования, нужно сделать так:

**SET enable\_hashjoin = off;**

Чтобы запретить планировщику использовать метод соединения слиянием, нужно сделать так:

**SET enable\_mergejoin = off;**

* для того чтобы запретить планировщику использовать соединение методом вло-женного 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 планиров-щик 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».

305

*Глава 10. Повышение производительности*

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 стои-мости выполнения запроса будут значительно выше, а вывод результирующих строк начнется значительно позднее, чем при использовании метода соединения слияни-ем. На это указывает значение параметра cost для верхнего узла дерева плана:

cost=226400.55..229014.87

* 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 результи-рующие строки не выводятся.

Сначала разрешим планировщику использовать метод соединения слиянием:

**SET enable\_mergejoin = on;**

SET

306

*10.4. Управление планировщиком*

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 —

* 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 уз-ла 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 внеш-нему 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 си-стеме используются буферизация и кэширование, а также с течением времени может

307

*Глава 10. Повышение производительности*

изменяться фактическая нагрузка на сервер. Поэтому время выполнения повторно-го 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 выполня-ется с помощью последовательного сканирования и сортировки. Обратите внимание, что при включении опции ANALYZE может выводиться дополнительная информация

308

*10.4. Управление планировщиком*

* фактически использовавшихся методах, о затратах памяти и др. В частности, сказа-но, что была использована внешняя сортировка на диске (Sort Method), приведены затраты памяти на ее выполнение, приведено число строк, удаленных при проверке условия их отбора (Rows Removed by Filter).

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 72 647, а оценки — 75 126. Это хороший результат работы планировщика.

Обратимся еще раз к запросу, который мы уже рассматривали выше, и выполним его

* опциями ANALYZE и COSTS OFF (для сокращения вывода). В плане этого запроса нас будет интересовать фактический параметр loops.

**EXPLAIN (ANALYZE, COSTS OFF)**

**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 (actual time=3.423..3.666 rows=426 loops=1)

Sort Key: s.seat\_no

Sort Method: quicksort Memory: 46kB

-> Nested Loop (actual time=0.236..0.993 rows=426 loops=1)

-> Seq Scan on aircrafts a (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 (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

(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)

309

*Глава 10. Повышение производительности*

Как видно из плана, значение параметра loops для узла, выполняющего сканиро-вание таблицы seats по индексу с построением битовой карты, равно трем. Это объясняется тем, что из таблицы aircrafts были фактически выбраны три стро-ки, и *для каждой* из них выполняется поиск в таблице seats. Для подсчета общих затрат времени на выполнение операций сканирования по индексу за три цикла нуж-но значение параметра actual time умножить на значение параметра loops. Таким образом, для узла дерева плана Bitmap Index Scan получим 0,064 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 па-мяти с использованием метода quicksort:

Sort Method: quicksort Memory: 46kB

До сих пор мы рассматривали только выборки, т. е. такие запросы, которые не изме-няют хранимых данных. Однако, кроме выборок, есть такие операции, как вставка, обновление и удаление строк. Нужно помнить, что хотя результаты выборки и не вы-водятся, тем не менее, она фактически все равно выполняется. Поэтому если требу-ется исследовать план выполнения запроса, модифицирующего данные, то для того, чтобы изменения на самом деле произведены не были, нужно воспользоваться тран-закцией с откатом изменений.

**BEGIN;**

BEGIN

**EXPLAIN (ANALYZE, COSTS OFF)**

**UPDATE aircrafts**

**SET range = range + 100**

**WHERE model ~ '^Air';**

QUERY PLAN

-----------------------------------------------------------------------

Update on aircrafts (actual time=0.299..0.299 rows=0 loops=1)

-> Seq Scan on aircrafts (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)

310

*10.5. Оптимизация запросов*

**ROLLBACK;**

ROLLBACK

* документации приводится важное предостережение о том, что нельзя экстраполи-ровать, т. е. распространять, пусть даже и с некоторыми поправками, оценки, полу-ченные для таблиц небольшого размера, на таблицы большого размера. Это объясня-ется 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 последова-тельное сканирование, а для большой — сканирование по индексу.

**10.5. Оптимизация запросов**

Мы рассмотрели базовые способы получения плана выполнения запроса и познако-мились с типичными компонентами плана. Эти знания призваны помочь в тех си-туациях, когда необходимо ускорить выполнение запроса. При принятии решения

* том, что выполнение какого-либо запроса нужно оптимизировать, следует учиты-вать 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.
* 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 рас-смотрим некоторые из них:

– обновление статистики, на основе которой планировщик строит планы;

– изменение исходного кода запроса;

– изменение схемы данных, связанное с денормализацией: создание материализо-ванных представлений и временных таблиц, создание индексов, использование вычисляемых столбцов таблиц;

311

*Глава 10. Повышение производительности*

– изменение параметров планировщика, управляющих выбором порядка соедине-ния наборов строк: использование общих табличных выражений (запросы с пред-ложением 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 досту-па к данным (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 опе-раций: агрегирование на основе хеширования (enable\_hashagg), материализа-ция временных наборов строк (enable\_material), выполнение явной сортиров-ки при наличии других возможностей (enable\_sort).

Необходимым условием для того, чтобы планировщик выбрал правильный план, яв-ляется наличие актуальной статистики. Если вы предполагаете, что планировщик опирается на неактуальную статистику, можно ее принудительно обновить с помо-щью команды ANALYZE. Например, обновить статистику для таблицы aircrafts можно, выполнив команду

**ANALYZE aircrafts;**

ANALYZE

* качестве примера ситуации, в которой оптимизация запроса представляется обос-нованной, рассмотрим следующую задачу. Предположим, что необходимо опреде-лить 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 озна-чает, 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 табли-цы «Бронирования» (bookings), нужно подсчитать соответствующие строки в таб-лице «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 подзапро-сом в предложении SELECT. Но это еще не окончательное решение. Теперь нужно сгруппировать полученный набор строк по значениям числа оформленных билетов.

312

*10.5. Оптимизация запросов*

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 плане получены очень большие оценки общей стоимости выполнения запроса:

cost=14000017.12..27994373.35

Универсальной зависимости между оценкой стоимости и реальным временем вы-полнения запроса не существует. Не всегда можно даже приблизительно предполо-жить, в какие затраты времени выльется та или иная оценка стоимости. Но, тем не менее, при рассмотрении других запросов оценок такого порядка нам еще не встре-чалось. Планировщик предполагает, что из таблицы tickets в подзапросе будет извлекаться всего по две строки, и эту операцию нужно будет проделать 1 314 раз: столько строк предположительно будет выбрано из таблицы bookings. Как видно

313

*Глава 10. Повышение производительности*

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 клавиш <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 ко-тором 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)

314

*10.5. Оптимизация запросов*

Теперь планировщик использует индекс для поиска в таблице tickets. Причем это поиск исключительно по индексу (Index Only Scan), поскольку нас интересует толь-ко 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)

315

*Глава 10. Повышение производительности*

-> 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)

* данном плане используется соединение хешированием (Hash Join). При этом ин-декс по таблице tickets игнорируется: таблица просматривается последовательно (Seq Scan on tickets t). Время выполнения модифицированного запроса оказыва-ется 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 модифициро-вать сам запрос, даже не создавая индекса.

Другие методы оптимизации выполнения запросов представлены в разделе «Кон-трольные вопросы и задания». Рекомендуем вам самостоятельно с ними ознакомить-ся и поэкспериментировать.

316

*Контрольные вопросы и задания*

**Контрольные вопросы и задания**

Перед выполнением упражнений нужно восстановить измененные значения пара-метров:

**SET enable\_hashjoin = on;**

SET

**SET enable\_nestloop = on;**

SET

1. Как вы думаете, почему при сканировании по индексу оценка стоимости ресур-сов, требующихся для выдачи первых результатов, не равна нулю, хотя исполь-зуется 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)

1. Как вы думаете, если в запросе присутствует предложение ORDER BY, и создан индекс по тем столбцам, которые фигурируют в предложении ORDER BY, то все-гда ли будет использоваться этот индекс или нет? Почему? Проверьте ваши предположения с помощью команды EXPLAIN.
2. Самостоятельно выполните команду EXPLAIN для запроса, содержащего общее табличное выражение (CTE). Посмотрите, на каком уровне находится узел пла-на, отвечающий за это выражение, как он оформляется. Учтите, что общие таб-личные выражения всегда материализуются, т. е. вычисляются однократно и результат их вычисления сохраняется в памяти, а затем все последующие об-ращения в рамках запроса направляются уже к этому материализованному ре-зультату.
3. Прокомментируйте следующий план, попробуйте объяснить значения всех его узлов и параметров.

317

*Глава 10. Повышение производительности*

**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)

1. В подавляющем большинстве городов только один аэропорт, но есть и такие го-рода, 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 ска-нирование 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 парамет-ре cost для узла Seq Scan, равная 3,04, не совпадает с первой оценкой стоимо-сти в параметре cost для узла HashAggregate?

318

*Контрольные вопросы и задания*

1. Выполните команду EXPLAIN для запроса, в котором использована какая-нибудь из оконных функций. Найдите в плане выполнения запроса узел с име-нем WindowAgg. Попробуйте объяснить, почему он занимает именно этот уро-вень в плане.
2. Проанализируйте план выполнения операций вставки и удаления строк. При-чем сделайте это таким образом, чтобы данные в таблицах фактически измене-ны не были.

8.\* Замена коррелированного подзапроса соединением таблиц является одним из способов повышения производительности.

Предположим, что мы задались вопросом: сколько маршрутов обслуживают са-молеты каждого типа? При этом нужно учитывать, что может иметь место такая ситуация, когда самолеты какого-либо типа не обслуживают ни одного марш-рута. 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;**

* в этом варианте коррелированный подзапрос раскрыт и заменен внешним соединением:

**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;**

319

*Глава 10. Повышение производительности*

Причина использования внешнего соединения в том, что может найтись мо-дель самолета, не обслуживающая ни одного маршрута, и если не использовать внешнее соединение, она вообще не попадет в результирующую выборку.

Исследуйте планы выполнения обоих запросов. Попытайтесь найти объясне-ние различиям в эффективности их выполнения. Чтобы получить усредненную картину, выполните каждый запрос несколько раз. Поскольку таблицы, участ-вующие в запросах, небольшие, то различие по абсолютным затратам времени выполнения 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». Про-ведите необходимые эксперименты с вашими запросами.

1. Одним из способов повышения производительности является изменение схе-мы данных, связанное с денормализацией, а именно: создание материализо-ванных представлений. В главе 5 было описано такое материализованное пред-ставление — «Маршруты» (routes). Команда для его создания была приведена в главе 6.

Проведите эксперимент: сначала выполните выборку из готового представле-ния, 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 обра-зом: сначала сохраните его в текстовом файле, а затем выполните с помощью команды ni утилиты psql.

Вы увидите, что затраты времени отличаются практически на два порядка. Ко-нечно, нужно помнить, что материализованные представления необходимо пе-риодически обновлять, чтобы их содержимое было актуальным.

10.\* Одним из способов повышения производительности является изменение схе-мы данных, связанное с денормализацией, а именно: использование вычисля-емых столбцов. Для примера рассмотрим таблицу «Бронирования» (bookings).

* ней столбец «Полная сумма бронирования» (total\_amount) является вычис-ляемым. Мы не будем сейчас говорить о том, каким образом его значения син-хронизируются с данными в таблице «Перелеты» (ticket\_flifgts), а лишь

320

*Контрольные вопросы и задания*

рассмотрим два запроса, возвращающие полные суммы бронирований. Пред-положим, 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, подтвержда-ющие 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 та-ком 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, поче-му план, сформированный для получения даже простой выборки из представ-ления, многоуровневый?

**EXPLAIN ANALYZE**

**SELECT \* FROM flights\_v;**

**EXPLAIN ANALYZE**

**SELECT \* FROM flights\_tt;**

321

*Глава 10. Повышение производительности*

Выполните более сложные запросы к представлению и временной таблице и сравните полученные результаты. Чтобы увидеть фактические затраты време-ни, включайте в команду EXPLAIN опцию ANALYZE.

Подумайте, при выполнении каких запросов к базе данных «Авиаперевозки» было бы целесообразно создать временную таблицу. Создайте ее и проведите эксперименты, подтверждающие эффективность ее использования.

1. Одним из способов повышения производительности является изменение схемы данных, связанное с денормализацией, а именно: создание индексов. Выполните следующий простой запрос к таблице «Билеты»:

**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 резуль-таты.

Предложите какой-нибудь запрос к базе данных «Авиаперевозки», для выпол-нения которого было бы целесообразно создать индекс. Создайте индекс и по-вторите запрос. Изучите полученный план, посмотрите, используется ли индекс планировщиком.

13.\* В самом конце главы мы выполняли оптимизацию запроса путем создания ин-декса 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;**

322

*Контрольные вопросы и задания*

Мы экспериментировали с параметрами планировщика enable\_hashjoin и enable\_nestloop при наличии индекса по таблице tickets.

**SET enable\_hashjoin = off;**

**SET enable\_nestloop = off;**

Однако полученные планы детально рассмотрены не были.

**Задание.** Проанализируйте эти планы. Посмотрите, в каких случаях использу-ются и в каких не используются индексы по таблицам bookings и tickets. Вспомните о таком понятии, как *селективность*, т. е. доля строк, выбираемых из таблицы.

1. В столбцах таблиц могут содержаться значения NULL. При сортировке строк по значениям таких столбцов СУБД по умолчанию ведет себя так, как будто зна-чение NULL превосходит по величине любые другие значения. В результате по-лучается, что если задан возрастающий порядок сортировки, то значения NULL будут идти последними, если же порядок сортировки убывающий, тогда они бу-дут первыми. Принимая решение о создании индексов, нужно учитывать требу-емый 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 таб-лицей 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:

323

*Глава 10. Повышение производительности*

**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 пред-ложением 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;**

324

*Контрольные вопросы и задания*

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 таблицыnulls. Не выполняя эту команду, попытайтесь ответить, будет ли использо-ваться созданный нами индекс при выполнении такой выборки, а затем про-верьте 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, чтобыон использовался при выполнении следующей выборки:

**SELECT \***

**FROM nulls**

**ORDER BY num NULLS FIRST;**

**Задание 3.** Выполните аналогичные эксперименты, задавая убывающий по-рядок 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 посмот-рите, каким будет фактическое время выполнения команд. За дополнительной информацией обратитесь к описанию команды CREATE INDEX, приведенному

в документации.

1. 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 тек-сте главы, однако они должны быть вам интуитивно понятны.

325

*Глава 10. Повышение производительности*

1. В разделе документации 19.7 «Планирование запросов» приведены парамет-ры, с помощью которых можно влиять на решения, принимаемые планиров-щиком. 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 за-прета 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 enable\_mergejoin = off;**

**SET enable\_nestloop = off;**

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 повы-шение оценок происходит именно в узле, отвечающем за соединение наборов строк. Эти оценки не означают, что время выполнения запроса будет стремить-ся к бесконечности. С помощью команды EXPLAIN ANALYZE выполните запрос и убедитесь в этом сами.

326

*Контрольные вопросы и задания*

**Задание.** Самостоятельно ознакомьтесь с содержанием раздела документации19.7 «Планирование запросов», а также раздела 14.3 «Управление планировщи-ком с помощью явных предложений JOIN» и проведите эксперименты с запро-сами, 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* нуж-но изменять значения конкретных параметров, правильно оценивать степень

и направленность их влияния, понимать взаимосвязь параметров, требуется опыт и изучение документации.

1. Самостоятельно ознакомьтесь с разделом документации 14.2 «Статистика, ис-пользуемая планировщиком».
2. Команда EXPLAIN имеет опцию BUFFERS. Ознакомьтесь с ней самостоятельно по разделу документации 14.1 «Использование EXPLAIN».
3. 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 за-трат ресурсов. Для повышения производительности СУБД в подобных ситуаци-ях 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 доку-ментации 14.4 «Наполнение базы данных». Смоделируйте ситуации, описанные в этом разделе документации, и выполните рекомендуемые действия.

327

**Рекомендуемые источники**

1. Гарсиа-Молина Г., Ульман Д. Д., Уидом Д. Системы баз данных. Полный курс: пер.

с англ. — М.: Вильямс, 2003. — 1088 с.

1. Грофф Дж. Р., Вайнберг П. Н., Оппель Э. Дж. SQL. Полное руководство: пер. с англ. —

3-е изд. — М.: Вильямс, 2015. — 960 с.

1. Дейт К. Дж. Введение в системы баз данных: пер. с англ. — 8-е изд. — М.: Вильямс, 2005. — 1328 с.
2. Новиков Б., Домбровская Г. Настройка приложений баз данных. — СПб.: БХВ-Петербург, 2012. — 240 с.
3. Новиков Б., Горшкова Е., Графеева Н. Основы технологий баз данных. — М.: Postgres Professional, 2018. — 182 с.
4. Селко Д. Стиль программирования Джо Селко на SQL: пер. с англ. — М.: Русская

редакция; СПб.: Питер, 2006. — 206 с.

1. Официальный сайт PostgreSQL <http://www.postgresql.org>
2. Postgres Professional <http://postgrespro.ru>
3. Учебные курсы Postgres Professional <http://postgrespro.ru/education/courses>

328

**Предметный указатель**

**A**

Aggregate, узел плана 301 ALL 205

ALTER MATERIALIZED VIEW 141 ALTER TABLE 117–122, 212, 220 ALTER VIEW 141 ANALYZE 312

ANY 205

ARRAY, литерал 66

array\_append, функция 65

array\_cat, функция 90

array\_length, функция 169, 195

array\_position, функция 67

array\_prepend, функция 65

array\_remove, функция 66

AS 152–153

AT TIME ZONE 127

avg, функция 168

**B**

BEGIN 259

BETWEEN 148

bigint, тип 51

bigserial, тип 54

Bitmap Scan, узел плана 300 boolean, тип 63, 87–88

**C**

CASCADE 116

CASE, условное выражение 152, 237 char, тип 54

character varying, тип 54 character, тип 54 CHECK 96

COALESCE, условное выражение 193

COMMENT 108

COMMIT 263

CONSTRAINT 101

COPY 217–218, 227

count, функция 46, 173, 195, 207 CREATE INDEX 243

CREATE MATERIALIZED VIEW 128 CREATE TABLE 27, 32, 107 CREATE TEMP TABLE 211 CREATE VIEW 123

CROSS JOIN 159, 237 current\_date, функция 57 current\_schema, функция 133 current\_time, функция 59 current\_timestamp, функция 62

**D**

date, тип 56

datestyle, параметр 79–82 date\_trunc, функция 62, 85 decimal, тип 52 DEFAULT 54, 96, 133–135 default\_transaction\_isolation,

параметр 259

DELETE 41, 49, 224–226

USING 225

DISTINCT 150, 176, 179, 220

double precision, тип 52, 74–76

DROP INDEX 245

DROP MATERIALIZED VIEW 130 DROP TABLE 37, 115 DROP VIEW 124, 128

329

*Предметный указатель*

**E**

END 263

EXCEPT 167

EXISTS 179

EXPLAIN 296–297

ANALYZE 306 extract, функция 62, 86

**F**

false, значение 63

FILTER 200

first\_value, функция 174

float, тип 53

FOR SHARE 286

FOR UPDATE 278

FOREIGN KEY 43, 100–104, 115, 121 FROM 38, 176, 181 FULL OUTER JOIN 162

**G**

GREATEST, условное выражение 193 GROUP BY 46, 168, 176

**H**

Hash Join, узел плана 303–304 HAVING 170, 176, 183

**I**

IF EXISTS 117

IN 177, 229

Index Only Scan, узел плана 300

Index Scan, узел плана 299–300

Infinity, значение 53, 75

INSERT 37–38, 213–217

ON CONFLICT 215–217, 227 integer, тип 51

INTERSECT 166 interval, тип 61, 84–85 intervalstyle, параметр 85

**J**

JOIN 152

json, тип 68

jsonb, тип 68

jsonb\_set, функция 72, 92

**L**

LEAST, условное выражение 193

LEFT OUTER JOIN 160 left, функция 210 length, функция 210 LIKE 146, 192, 254

LIMIT 151, 176, 234, 246, 297 LOCK TABLE 279 lower, функция 248

**M**

max, функция 168

Merge Join, узел плана 304–305 min, функция 168

**N**

NaN, значение 53, 74, 76

Nested Loop, узел плана 302–303 NOT NULL 97

NULL, значение 18, 103, 136 отличие от пустой строки 138 при сортировке 247, 323

NULLIF, условное выражение 193 numeric, тип 52, 73–74

**O**

OFFSET 151, 176, 324

ON CONFLICT 215–217, 227

ON DELETE

CASCADE 102

NO ACTION 102

RESTRICT 102

SET DEFAULT 103

SET NULL 103

330

ON UPDATE

CASCADE 104

ORDER BY 39, 46, 48, 149, 176, 234, 246 OVER 173

**P**

pg\_dump 34

PRIMARY KEY 34–44, 99, 107–115, 242 psql 26–27, 32–37

автодополнение 107

выход 29

завершение комады SQL 35 описание индекса 242, 252 описание представления 127 описание таблицы 36, 107 расширенный формат 127 редактирование буфера 35 секундомер 243 список индексов 245 список схем 131 список таблиц 27 способ ввода команд 34 справка 27, 32

терминал в Windows 26 public, схема 132

**R**

rank, функция 173, 225, 232 Read Committed, уровень

изоляции 258, 261–265, 275, 280–283

особенности UPDATE 280–283 Read Uncommitted, уровень

изоляции 258–261 real, тип 52, 74–76 REFERENCES 101

REFRESH MATERIALIZED VIEW 129 Repeatable Read, уровень

изоляции 258, 265–269 RETURNING 213, 219, 224, 226

*Предметный указатель*

RIGHT OUTER JOIN 161 right, функция 210 ROLLBACK 261 round, функция 149

**S**

search\_path, параметр 105, 131 SELECT 38, 145–210

Seq Scan, узел плана 296–299

serial, тип 53, 76–78

Serializable, уровень изоляции 258,

269–275, 286–291 SET 80, 132, 305 SHOW 79, 132

SIMILAR TO 193

smallint, тип 51

smallserial, тип 54

Sort, узел плана 298, 309

string\_agg, функция 182

strpos, функция 208

substr, функция 208

sum, функция 206

**T**

text, тип 54

time, тип 58, 78

with time zone (timetz) 58 timestamp, тип 60

with time zone (timestamptz) 60 to\_char, функция 57, 82 true, значение 63

TRUNCATE 226

**U**

UNION 166, 188 UNIQUE 98, 110–122 unnest, функция 68, 196

331

UPDATE 40, 48, 219–223 FROM 223

особенности на уровне Read

Committed 280–283

**V**

VALUES 164, 176

varchar, тип 54

**А**

Аномалия сериализации 257 Атомарность, свойство

транзакции 256, 277 Атрибут 16

**Б**

Блокировка 256, 278–279

на уровне строк 278, 285

на уровне таблицы 279

предикатная 286

**Г**

Группировка 46, 168

«Грязное» чтение, аномалия 257, 259

**Д**

Демо-база «Авиаперевозки»

описание 19–21, 125, 129, 156, 189

схема 22

установка 27

функция bookings.now 207, 229, 276

Демо-база «Успеваемость» 14, 16, 95, 104

Денормализация 138, 320–322 Долговечность, свойство

транзакции 256, 277

**З**

Значение по умолчанию 54, 96, 103, 133–135

**W**

WHERE 40, 145, 176

WINDOW 174

WITH 186, 201–203

WITH ORDINALITY 198

**И**

Изолированность, свойство транзакции 256, 277

Индекс 36, 241–254, 286

метод доступа 293–294

описание в psql 242, 252

по выражению 248

по нескольким столбцам 246

список в psql 245

уникальный 242, 247

ускорение запроса 244, 314

частичный 248, 253

**К**

Карта видимости 294

Класс оператора 254

Ключ

альтернативный 17

внешний 17, 43–44, 100, 104, 115, 121

естественный 33, 42, 110 первичный 17, 34–44, 99, 107–115 потенциальный 17 составной 42, 113 суррогатный 33, 53, 109 уникальный 98, 110–122

Колонка (столбец) 15 Комментарии в SQL 106 Кортеж 16

332

**М**

Метод доступа 293–294 исключительно по индексу 294,

300

по битовой карте 294, 300 по индексу 293, 299–300 последовательный 293, 296–299

Многоверсионность 256 Модель данных

реляционная 13

**Н**

Неповторяющееся чтение, аномалия 257, 263–268

**О**

Общее табличное

выражение 186–192, 233, 317

рекурсивное 187–189, 201–203

* командой DELETE 224
* командой INSERT 213–215
* командой UPDATE 219 Ограничение целостности 16, 96–104,

255

CHECK 33, 96, 109–111, 113, 118, 135

NOT NULL 33, 97, 118, 135 индексная поддержка 36, 98–99,

242 каскадное обновление 104, 137 каскадное удаление 43, 102 каскадное удаление

объектов 116, 141 первичный ключ 17, 99, 242 ссылочное (внешний ключ) 17,

43–44, 100–104, 115, 121 уникальный ключ 98, 136, 215,

242

*Предметный указатель*

Операторы SQL

DDL 95–144

DML 145–239

классификация 18

Отношение 15

как результат реляционных

операций 156 кардинальное число 16 степень 16

**П**

План выполнения запроса 155, 296 оценка стоимости 297, 317, 326 оценка числа строк 297 расхождение оценок и факта 315 фактические затраты 307

Планировщик 155, 241, 296 управление 305

Подзапрос 177–192

* предложении FROM 181–183
* предложении HAVING 183–184
* предложении SELECT 180–181
* предложении WHERE 177–180, 205

вложенный 184–185, 191, 230 коррелированный 179, 185, 230,

315, 319

некоррелированный 177

общее табличное

выражение 186–192, 233, 317 рекурсивный 187–189, 201–203 скалярный 177, 188

Последовательность 54, 111 Потерянное обновление,

аномалия 257, 261–263, 283–284

333

*Предметный указатель*

Представление 45, 123–131, 234 вертикальное и

горизонтальное 142 материализованное 128, 142, 189,

320

обновляемое 141

описание в psql 127

применение 130 Процессор языка запросов 15 Псевдоним 153, 157

**Р**

Регулярные выражения

POSIX 147, 254

SQL 193

**С**

Селективность выборки 246, 251, 293 Сериализация 257

ошибки 265, 268–275, 286 Система управления базами данных

(СУБД) 14

установка 25

Снимок данных 256

Согласованность, свойство

транзакции 256, 277 Соединение 152, 294, 315, 319

без использования JOIN 155

* команде DELETE 225
* команде UPDATE 223 внешнее 159–162, 189

левое 160 полное 162 правое 161

декартово 159, 237

на основе неравенства 158 таблицы с собой 156, 158 эквисоединение 153, 295

Сортировка 39, 46, 48, 149, 241–247, 295, 298, 323

* памяти и внешняя 309 Способ соединения 294–295

вложенный цикл 295, 302–303 слиянием 295, 304–305 хешированием 295, 303–304

Статистика 297, 312 Столбец (колонка) 15

вычисляемый 148, 320 Страница 293 Строка 15

версия 256 Схема 36, 108, 131–133

public 132

путь поиска 105, 131 список в psql 131

**Т**

Таблица 15

виртуальная 187, 202, 236

виртуальная (VALUES) 164

временная 211, 220, 321

использование для

журналирования 211

описание в psql 36, 107

связь 1:1 114, 162

список в psql 27

ссылающаяся и ссылочная 17, 43,

100, 106, 115, 121

Тестовые данные, генерация 235

Типы данных, приведение 44, 57, 83,

119, 237

Типы данных: JSON 68, 92–93, 112, 143, 210

вхождение 70

значение по ключу 71, 92 конкатенация 71, 93 проверка ключа 71 удаление ключа 93

334

Типы данных: даты и времени 56, 78–87, 191

арифметические операции 84 временные отметки 60, 110, 112,

127

время суток 58

дата 56

интервалы 61

формат ввода и вывода 79–82, 85 часовой пояс 58, 61, 110, 112, 127,

150

Типы данных: логический 63, 87–88

Типы данных: массивы 44, 64, 90–92, 169, 192, 195

вхождение 67, 196 конкатенация 65, 90 многомерные 92 пересечение 67 преобразование в таблицы 68,

196

срезы 66

*Предметный указатель*

Типы данных: символьные 33, 42, 54, 112, 199, 237

использование кавычек 55–56 конкатенация 237 пустые строки 138

Типы данных: числовые 33, 51

* плавающей точкой 52, 74, 108 фиксированной точности 52, 73,

112–113 целочисленные 51, 114, 118 целочисленные с

автоинкрементом 53, 76–78, 109, 111

Транзакция 18, 255–291 Триггер 276

**Ф**

Фантомное чтение, аномалия 257,

265–268, 284–285 Функция

агрегатная 46, 168–170, 182, 200, 301

оконная 170–175, 200, 232, 319

335

Моргунов Евгений Павлович

**POSTGRESQL. ОСНОВЫ ЯЗЫКА SQL**

*Учебноое пособие*

При поддержке Postgres Professional

<http://postgrespro.ru>

Редакторы *Е. В. Рогов, П. В. Лузанов*

Подписано в печать 31.10.18.

Формат 70 1001/16. Печать офсетная. Усл. печ. л. 27,09.

Тираж 1000 экз. Заказ №7782.

«БХВ-Петербург», 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета

ООО «Принт-М», 142300, М. О., г. Чехов, ул. Полиграфистов, д. 1