PROGRAMMER TO PROGRAMMER™

ОСНОВЫ PostgreSQL

Ричард Стоунз,
Нейл Мэттью

wrox
PROGRAMMER TO PROGRAMMER™

СИМВОЛ

Beginning Databases with PostgreSQL

Richard Stones and Neil Matthew

PostgreSQL
Основы

Ричард Стоунз, Нейл Мэттью

Санкт-Петербург
2002

Ричард Стоунз, Нейл Мэттью

PostgreSQL. Основы

Перевод П. Шера

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Науч. редактор	*А. Тенихин*
Редактор	*В. Овчинников*
Корректура	*С. Беляева*
Верстка	*А. Дорошенко*

Стоунз Р., Мэтью Н.
PostgreSQL. Основы. – Пер. с англ. – СПб: Символ-Плюс, 2002. – 640 с., ил.
ISBN 5-93286-043-X

PostgreSQL – реляционная СУБД с открытым исходным кодом – в последнее время стремительно завоевывает популярность. И это не случайно. Корнями уходя в академическую среду, она усилиями команды разработчиков, объединяющей талантливых людей со всего мира, попала в Интернет. Многие малые и средние предприятия переносят бизнес с запатентованных баз данных на PostgreSQL, и это неоспоримо свидетельствует о ее успехе.

Читатели постепенно пройдут от основ к проектированию и созданию баз данных, научатся интегрировать их с языками программирования, предназначенными для Интернета. Книга рассчитана на новичков, которым, однако, не помешает знакомство с SQL, а в некоторых главах – с PHP, Perl и Java.

Рассмотрены инсталляция из двоичного дистрибутива и исходных текстов для UNIX и Windows, работа с графическими средствами, различные формы запросов, обобщенные функции и объединения, агрегаты, транзакции, блокировки, хранимые процедуры и триггеры. А кроме того, контроль за производительностью, настройка и управление сервером, установление соединения и выполнение SQL-операторов с помощью C (libpq) и встроенного SQL, разработка приложений на PHP, Perl и Java.

ISBN 5-93286-043-X
ISBN 1-861005-15-6 (англ)

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 16.08.2002. Формат 70×100 1/16 . Печать офсетная.
Объем 40 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Благодарности авторов

Мы, Ричард и Нил, хотели бы поблагодарить свои семьи:

Жену Рика, Энн, и его детей, Дженни и Эндрю, за их терпение теми длинными вечерами и выходными, когда он писал эту книгу. Рик также хотел бы поблагодарить их за то, что они проявили такое понимание, когда было решено увеличить объем книги.

Жену Нила, Кристин, за ее неизменную поддержку и понимание и его детей, Александру и Эдриена, за то, что они думают, что здорово иметь папу, который умеет писать книжки.

Мы также хотели бы поблагодарить всех тех, кто сделал возможным появление этой книги.

В первую очередь всех, кому понравились наши предыдущие книги, «Beginning Linux Programming» (Начала программирования в Linux) и «Professional Linux Programming» (Программирование в Linux для профессионалов). Благодаря вам они имели такой успех, и именно ваши отклики побудили нас взяться за еще одну книгу.

Хотелось бы поблагодарить команду Wrox за их нелегкую работу над книгой, особенно Дану М. (Dan M.), которая очень помогла нам с исходными требованиями. Спасибо за ее работу на начальном этапе написания книги, а также спасибо труженикам из команды в Мумбае (Бомбее), особенно Дилипу Т. (Dilip T), Манжу (Manju), Инду Б. (Indu B), Нилешу (Nilesh) и Вижаю Т. (Vijay T), а также всем остальным, работавшим за кулисами.

Также хотелось бы поблагодарить тех, кто внес в книгу дополнительные материалы; это было великолепно!

Особая благодарность команде рецензентов, от которых мы получили замечания и предложения высочайшего качества. То, что они сделали для улучшения книги, далеко выходит за рамки их служебного долга. Огромное спасибо всем и каждому. Если в книге остались какие-то ошибки, это, конечно же, целиком наша вина.

Хотелось бы также поблагодарить наших работодателей, GEHE, за их поддержку при написании этой книги.

Спасибо всем тем многим людям, которые потратили свои силы и время на то, чтобы сделать PostgreSQL замечательной свободно распространяемой базой данных; желаем им и их великолепному продукту всяческих успехов в будущем.

Хотелось бы отдать должное Линусу (Linus) за платформу Linux, RMS – за прекрасный набор программ GNU и GPL, а также постоянно расширяющемуся сообществу невоспетых героев, которые решили сделать свое программное обеспечение бесплатным, не забывая и тех, кто продолжает другими способами содействовать делу Open Source и GPL-лицензий программного обеспечения.

Предисловие

Как произносить название «PostgreSQL»

Прежде чем приступать к постижению основ какой-либо новой технологии, необходимо научиться правильно произносить ее название. А так как вы собираетесь изучать PostgreSQL (не самое простое для произношения название), некоторая осторожность не повредит.

Изначально система управления базами данных (СУБД) называлась Postgres. Когда СУБД стала поддерживать язык SQL, и к разработке Postgres присоединилось Интернет-сообщество, в конец названия была логичным образом добавлена частица «SQL».

Название произносится «пост-грес-ку-эл(ь)» (распространенный вариант произношения «постгре-эс-ку-эл(ь)» является ошибочным).

О чем рассказывается в этой книге

Книга состоит из 18 глав, скомбинированных так, чтобы познакомить читателя с основами PostgreSQL, а затем сделать шаг вперед, перейдя к основам программирования баз данных; будут рассмотрены модель данных PostgreSQL, разнообразные типы данных и расширяемая архитектура.

Глава 1 описывает базы данных в целом, а также кратко поясняет, чем они могут быть полезны. Рассказывается о месте PostgreSQL в общей картине.

В главе 2 более основательно рассматриваются принципы реляционных СУБД на базе того материала, который вы усвоили из главы 1 (об истории и архитектуре PostgreSQL). Кроме того, создается некая простая база данных и рассказывается о том, как можно использовать в ней стандартные типы данных.

Глава 3 проведет читателя по всем этапам инсталляции PostgreSQL из дистрибутива или исходных кодов как в среде UNIX, так и в операционной системе Windows. Представлены основы создания и заполнения таблиц.

Глава 4 содержит формальное рассмотрение основы языка SQL – оператора SELECT. Рассказано о многочисленных параметрах, с помощью которых можно сконфигурировать PostgreSQL так, чтобы она правильно интерпретировала данные и отображала нужную информацию.

В главе 5 рассмотрены такие альтернативы psql (для доступа и администрирования баз данных PostgreSQL), как pgAdmin и pgAccess. Показано, как использовать продукты Microsoft Office (Microsoft Excel и Microsoft Access) для обработки и представления данных.

Глава 6 представляет обзор других средств манипулирования данными от добавления новых данных при помощи команды INSERT до удаления всех строк таблицы командой TRUNCATE.

Глава 7 основана на материале главы 4 и охватывает наиболее сложные аспекты языка SQL: погружается в глубины оператора SELECT – к агрегатным функциям, внешним объединениям и т. д.

В главе 8 рассказано о более формальном подходе к обработке данных и о таких дополнительных возможностях, как создание ограничений.

Глава 9 полностью посвящена транзакциям, представлены правила ACID, уровни изоляции ANSI и реализация блокировок.

Глава 10, представляя операторы и функции, рассказывает о специфике PostgreSQL, расширении функциональности СУБД за счет хранимых процедур и загружаемого процедурного языка PL/pgSQL.

Повествование *главы 11* сосредоточено на администрировании баз данных. Охвачено управление сервером, обслуживание базы данных, резервное копирование, восстановление базы и т. д.

В главе 12 речь идет о проектировании базы данных. Подробно описаны все этапы генерирования схемы, традиционные шаблоны.

В главе 13 вниманию читателя предлагается создание собственных клиентских приложений на языке C.

Глава 14 использует материал предыдущей главы, рассказывая о другом, более переносимом способе объединения PostgreSQL с языком C – операторы SQL вводятся непосредственно в исходный код, затем с помощью транслятора ecpg генерируется код на языке C, код компилируется и на выходе получается рабочая программа.

В главе 15 рассматривается расширение PostgreSQL для поддержки PHP, а также способы доступа к базе данных PostgreSQL из языка PHP. Здесь же представлен абстрактный интерфейс PEAR для доступа к базам данных.

Глава 16 изучает доступ к базам данных PostgreSQL из Perl. Рассматриваются мощные средства манипулирования строками, имеющиеся в Perl, модули Perl DBI и DBIx, которые облегчают программирование баз данных.

В главе 17 рассказывается о программах Java, использующих JDBC для доступа к реляционным базам данным PostgreSQL. Также кратко упоминаются новые возможности, предоставляемые версией 3.0 JDBC API.

В главе 18 предполагается, что читатели уже готовы самостоятельно продолжать исследования возможностей PostgreSQL. Чтобы помочь им на этом пути, приведена некоторая справочная информация.

Условные обозначения

В этой книге применяется несколько стилей текста и способов форматирования. Это сделано для того, чтобы различать разнородные сведения. Приведем примеры таких стилей:

Попробуйте сами

Этот раздел содержит практический пример.

Как это работает

Здесь объясняется, что происходит при выполнении данного примера.

Важная информация, ключевые моменты, дополнительные сведения выводятся таким вот образом, чтобы привлечь ваше внимание. Не пренебрегайте ими!

- Увидев что-то вроде `test.sql`, знайте, что это имя файла, название объекта или имя функции.
- Элементы интерфейса, например Control Panel, выделяются особым шрифтом.

Для кода применяется несколько шрифтов. Если что-то обсуждается в тексте главы, например утилита командной строки `psql`, то ее название выделяется особым шрифтом. Если присутствует блок кода, который можно ввести как программу и запустить, он помещается в серый прямоугольник:

```
SELECT * FROM item;
```

Можно встретить код, представленный несколькими стилями:

```
for(n = 0; n < nfields; n++) {
     if(PQgetisnull(result, r, n))
         printf(" %s is NULL,", PQfname(result, n));
      else
          printf(" %s = %s(%d),",
           PQfname(result, n),
           PQgetvalue(result, r, n),
           PQgetlength(result, r, n));
        }
```

Серым фоном выделен новый код или же код, имеющий непосредственное отношение к текущему обсуждению, при этом он показан в контексте уже рассмотренного ранее кода (теперь он на белом фоне).

Если приводится текст, набираемый в ответ на приглашение к вводу команды, он будет выделен следующим образом:

```
# SELECT * FROM item;
```

Выходные данные показаны тем же шрифтом, что и код:

```
 item_id |  description  | cost_price | sell_price
---------+---------------+------------+--------------
       1 | Wood Puzzle   |      15.23 |      21.95
       2 | Rubic Cube    |       7.45 |      11.49
       3 | Linux CD      |       1.99 |       2.49
       4 | Tissues       |       2.11 |       3.99
       5 | Picture Frame |       7.54 |       9.95
       6 | Fan Small     |       9.23 |      15.75
       7 | Fan Large     |      13.36 |      19.95
       8 | Toothbrush    |       0.75 |       1.45
       9 | Roman Coin    |       2.34 |       2.45
      10 | Carrier Bag   |       0.01 |       0.00
      11 | Speakers      |      19.73 |      25.32
(11 rows)
```

Получение исходного кода

Работая над примерами из этой книги, можно вводить тексты программ вручную. Многие читатели предпочитают поступать именно так, потому что это хорошая возможность познакомиться с описываемыми техниками программирования.

Как бы то ни было, исходный код, приведенный в данной книге, доступен в Интернете по адресу *http://www.wrox.com*.

Те, кто любит вводить текст с клавиатуры, смогут по этим файлам проверить результаты своих трудов. Если же вы не сторонник ручного набора, тогда загрузите исходные коды с веб-сайта. В любом случае файлы помогут при отладке и обновлениях.

Опечатки

Авторы приложили все усилия к тому, чтобы в тексте и исходном коде не было ошибок. Однако человеку свойственно ошибаться, и авторы считают, что необходимо информировать читателя о найденных и исправленных опечатках по мере их обнаружения.

Списки опечаток для всех наших книг выложены в Интернете по адресу *http://www.wrox.com*. Если вы найдете ошибку, которой нет в списке, пожалуйста, сообщите об этом авторам.

На указанном выше сайте сосредоточена и другая информация, в том числе исходные коды программ всех наших книг, избранные главы, анонсы будущих изданий, а также статьи и различные мнения по сходным темам.

Техническая поддержка

Если при чтении возникли затруднения и вы хотите проконсультироваться с экспертом, досконально знающим эту книгу, отправьте электронное письмо по адресу *support@wrox.com*, указав в поле «Subject» название книги и четыре последние цифры ISBN. Письмо попадет в группу технической поддержки, которая ведет архив часто задаваемых вопросов. Ответ, содержащий общую информацию, будет выслан ими немедленно. Эта группа также отвечает на общие вопросы, касающиеся книги и сайта.

Более серьезные вопросы пересылаются техническому редактору, ответственному за определенную книгу. Редакторы знакомы с основными языками программирования и конкретными продуктами, так что они могут ответить на детальные технические вопросы по теме книги. Если проблема разрешена и оказалось, что она вызвана наличием опечаток в тексте, редактор может разместить на сайте соответствующую информацию.

Наконец, представим себе маловероятную ситуацию – редактор не смог разрешить вашу проблему. Тогда запрос будет адресован автору. В принципе авторов стараются ограждать от всего, что может отвлечь их внимание от основного занятия. Однако вопросы, непосредственно относящиеся к книгам, обязательно будут им отправлены. Все авторы Wrox помогают осуществлять поддержку на сайте. Они могут сами отправить ответ читателю или же передать его редактору или в отдел технической поддержки, который перешлет ответ.

Форумы P2P в Интернете

Поддержка Wrox не заканчивается, когда перевернута последняя страница. Если у вас возник вопрос, выходящий за рамки данной книги, или вы изменили код, представленный в книге, в своих личных целях и теперь вам требуется техническая поддержка, приходите на форум P2P (*http://p2p.wrox.com/*).

P2P – это сообщество программистов, которые делятся проблемами и опытом. Множество рассылок охватывает все современные Интернет-технологии и методики программирования. Ссылки, сетевые ресурсы

и архивы предоставляют всеобъемлющую базу знаний, в которой найдется что-то интересное и для новичка и для опытного программиста.

Рассылки проверяются администратором сервера, который может удалять некоторые сообщения для того, чтобы гарантировать их соответствие теме обсуждения и корректность. Это не означает, что сообщения появляются в рассылках лишь после того, как они прочитаны и одобрены, просто таким образом предотвращается поток «макулатурной» почты. Права на чтение предоставляются всем желающим (анонимно), а для того чтобы отправить сообщение, нужно зарегистрироваться (по крайней мере указать свой адрес электронной почты).

Поделитесь своими впечатлениями

Связь с читателем не обрывается в тот момент, когда он выходит из книжного магазина. Авторы понимают, что ошибки в тексте могут испортить удовольствие от прочтения книги и привести к бесполезной потере времени, поэтому всеми силами стараются минимизировать урон, который они могут нанести.

Сообщите нам, насколько вам понравилась (или не понравилась) эта книга и что бы вы посоветовали изменить в следующий раз. Можете присылать комментарии по электронной почте на адрес *feedback@wrox.com*. Не забудьте указать в сообщении название книги.

1

Введение в PostgreSQL

Книга посвящена одному из наиболее успешных программных продуктов с открытыми исходными кодами (Open Source) – реляционной базе данных PostgreSQL. Как поклонники баз данных, так и разработчики Open Source приняли PostgreSQL «на ура». Применение базы данных удобно для создания любого приложения, манипулирующего значительным объемом данных, а PostgreSQL – это превосходно реализованная система управления базами данных (СУБД), она обладает всеми необходимыми возможностями, находится в свободном доступе и имеет открытые исходные коды.

PostgreSQL может использоваться практически из всех основных языков программирования, включая C, C++, Perl, Python, Java, Tcl и PHP. Она очень близко следует промышленному стандарту для языков запросов, SQL92. В 2000 году она получила премию Linux Journal Editor's Choice Award («Выбор редактора» журнала Linux Journal) как лучшая СУБД.

Но не будем забегать вперед. Читателям, вероятно, интересно, что же такое PostgreSQL и для чего эта СУБД может быть полезна.

Эта глава подготавливает к восприятию оставшейся части книги и дает некоторые предварительные сведения о базах данных в целом, различных типах баз данных, областях их применения и о том, как в эту картину вписывается PostgreSQL.

Программная обработка данных

Почти все нетривиальные компьютерные приложения оперируют большими объемами данных, и множество приложений написано глав-

ным образом для того, чтобы обрабатывать данные, а не совершать расчеты.

Некоторые авторы считают, что во всем мире 80% разработки приложений так или иначе связано с совокупностями данных, хранящимися в базах данных, поэтому СУБД составляют основу многих приложений. Средства для программного управления данными имеются в изобилии. Большинство хороших книг по программированию содержат главы о создании, хранении и обработке данных. Авторы этой книги тоже писали о программной обработке данных:

- «Основы программирования для Linux» («Beginning Linux Programming»), Нейл Мэттью и Ричард Стоунз, Wrox Press (ISBN 81-7366-156-1) рассказывает о DBM-библиотеке.
- «Профессиональное программирование для Linux» («Professional Linux Programming»), Нейл Мэттью и Ричард Стоунз, Wrox Press (ISBN 1-861003-01-3) содержит главы о системах управления реляционными базами данных PostgreSQL и MySQL.

Данные могут иметь любую форму и размер, а способ их обработки зависит от природы данных. В некоторых случаях данные могут быть представлены одним-единственным значением, например числом π, которое может использоваться в программе, рисующей окружности. Это значение может быть жестко задано в самом приложении для отношения длины окружности к ее диаметру. Данные такого типа называется константами, потому что они никогда не меняются.

В качестве еще одного примера постоянных данных рассмотрим кросс-курсы валют некоторых европейских государств. В странах Европейского союза, которые участвуют в создании единой европейской валюты (евро), величины курсов обмена валют зафиксированы до шестого знака после запятой.

Приложение-конвертор валют Европейского союза могло бы использовать жестко запрограммированную таблицу, в которой содержались бы названия валют и базовый валютный курс, количество национальных единиц валюты в одном евро. Эти курсы никогда не изменятся.

Однако таблица валют может разрастаться. Новые страны могут входить в Евросоюз, тогда курсы их национальных валют также будут зафиксированы и их придется внести в таблицу.

Получается, что конвертор требуется изменить, встроенная таблица изменилась и приложение должно быть перестроено. Такую операцию необходимо выполнять при каждом изменении таблицы валют.

Удобнее сделать так, чтобы приложение читало файл, содержащий некоторую простую информацию о валютах, например название, принятое интернациональное обозначение и курс обмена. Тогда можно будет при необходимости изменения таблицы модифицировать только файл, не изменяя само приложение.

Используемый файл с данными не имеет какой-то специальной структуры; это просто несколько строк текста, имеющих значение для конкретного приложения, читающего этот файл. Поскольку файлу не свойственна определенная структура, будем называть его «плоским» (flat). Пример файла с курсами валют приведен в табл. 1.1:

Таблица 1.1. Плоский файл с курсами валют

France	FRF	6.559570
Germany	DEM	1.955830
Italy	ITL	1936.270020
Belgium	BEF	40.339901

Базы данных, состоящие из плоских файлов

Плоские файлы могут быть очень полезны во многих видах приложений. До тех пор пока файл не настолько велик, чтобы его обработка стала слишком сложной, вполне можно использовать плоские файлы.

Многие системы и приложения, в том числе написанные для UNIX, используют такие файлы для хранения данных или обмена информацией. В качестве примера можно привести файл паролей UNIX.

Плоский файл, рассмотренный выше, состоит из некоторого количества информационных элементов или атрибутов, вместе они составляют то, что можно назвать «записью». Записи устроены так, что каждая строка файла является отдельной записью, а файл собирает связанные записи вместе. Бывают случаи, когда такая конструкция не совсем подходит для поддержания работы приложения, тогда приходится вводить дополнительные возможности.

Предположим, что в приложение из предыдущего раздела нужно для каждой страны внести язык, на котором в ней говорят, а также площадь и численность населения. В плоском файле одна строка соответствует одной записи, при этом каждая запись состоит из нескольких атрибутов. Каждый отдельный атрибут всегда находится внутри записи на одном и том же месте, например обозначение валюты всегда является вторым атрибутом, поэтому можно просматривать данные по столбцам, в столбце всегда представлена информация одного вида. Таким образом, добавление языка, на котором говорят в какой-то стране, можно понимать как добавление еще одного столбца во все имеющиеся строки.

Но при введении этого столбца возникает затруднение – оказалось, что в некоторых странах существует несколько официальных языков. Поэтому в записи для Бельгии должен присутствовать как французский, так и фламандский язык. А в запись о Швейцарии нужно внести четыре языка.

Такая проблема носит название **повторяющихся групп**. Ситуация выглядит следующим образом: абсолютно законный элемент (язык) может повторяться в записи несколько раз, поэтому повторяется не только запись (строка), но и данные в ней. Плоские файлы не справляются с задачей, ведь невозможно определить, где заканчивается перечисление языков и начинается следующая часть записи. Единственным способом выхода из положения является введение в файл некоторой структуры, но тогда он перестает быть плоским.

Проблема повторяющихся групп возникает очень часто, и именно она стимулировала разработку более сложных систем управления базами данных. Можно попытаться решить задачу, используя обычные текстовые файлы, в которые введена некоторая структура. Такие файлы часто тоже называют плоскими, но название «структурированный плоский файл» точнее. Рассмотрим новый пример.

Приложение, которое хранит информацию о DVD-дисках, должно помнить год выпуска, режиссера, жанр фильма и состав исполнителей. Можно создать для этих данных файл, немного напоминающий файл `.ini` в Windows:

```
[2001: A Space Odyssey]
year=1968
director=Stanley Kubrick
genre=science fiction
starring=Keir Dullea
starring=Leonard Rossiter
...
[Toy Story]
...
```

Проблема повторяющихся групп решена введением тегов, указывающих тип каждого элемента записи.

Теперь приложение должно прочитать и интерпретировать более сложный файл только для того, чтобы добраться до содержащихся в нем данных. Выполнять в такой структуре обновление записи и поиск не очень удобно. Как можно быть уверенным в том, что описание жанра или классификация выбраны из определенного подмножества? Как, не прилагая особых усилий, вывести упорядоченный список фильмов Кубрика?

По мере того как требования к данным становятся все более и более сложными, приходится писать все больше и больше кода для чтения и хранения этих данных. Если же расширить наше DVD-приложение, включив в него информацию, которая может быть полезна владельцу магазина, занимающегося выдачей фильмов напрокат, например данные о берущих диски и взимаемой плате, о возвратах и бронировании фильмов, то перспектива использовать плоский файл для хранения всей этой информации представляется не очень-то заманчивой.

Третья, также очень распространенная проблема – это размер. Хотя, применив грубую силу, можно организовать в описанной выше структуре сложный поиск по запросам типа «найти адреса всех клиентов, которые брали на прокат одну и более комедий за последние три месяца», но запрограммировать это нелегко, не говоря уже о том, что производительность будет крайне низкой. Дело в том, что для поиска любых данных приложение всегда просматривает весь файл целиком, даже если запрашивается только одно значение, например «Сколько фильмов было выпущено в 1968 году?».

Нужен некий универсальный способ хранения и извлечения данных, а не решение, придумываемое каждый раз для того, чтобы обойти несколько отличающиеся друг от друга, но весьма схожие проблемы, возникающие в большинстве систем обработки данных.

Нужна база данных.

Что такое база данных?

Доступный в Интернете словарь Merriam-Webster (*http://www.m-w.com*) определяет базу данных как большой набор данных, организованный специальным образом для обеспечения быстрого поиска и извлечения данных (например, с помощью компьютера).

Система управления базами данных (СУБД), как правило, представляет собой комплект библиотек, приложений и утилит, освобождающих разработчика приложения от груза забот, касающихся деталей хранения и управления данными. СУБД также предоставляет средства поиска и обновления записей.

За многие годы для решения различных видов проблем хранения данных было создано множество СУБД.

Типы баз данных

В 1960–70-х годах разрабатывались базы данных, которые тем или иным способом решали проблему повторяющихся групп. Эти методы привели к созданию **моделей** систем управления базами данных. Основой для таких моделей, используемых и по сей день, послужили исследования, проводимые в компании IBM.

Одним из основополагающих факторов проектирования ранних СУБД была эффективность. Гораздо легче манипулировать записями базы данных, имеющими фиксированную длину или, по крайней мере, фиксированное количество элементов в записи (столбцов в строке). Так удается избежать проблемы повторяющихся групп. Тот, кто программировал на каком-либо процедурном языке, без труда поймет, что в этом случае можно прочитать каждую запись базы данных в простую структуру C. Однако в реальной жизни такие удачные ситуа-

ции встречаются редко, поэтому программистам приходится обрабатывать не так удобно структурированные данные.

База данных с сетевой структурой

Сетевая модель вводит в базы данных указатели – записи, содержащие ссылки на другие записи. Так, можно хранить запись для каждого заказчика. Каждый заказчик в течение некоторого времени разместил у нас множество заказов. Данные расположены так, что запись заказчика содержит указатель ровно на одну запись заказа. Каждая запись заказа содержит как данные по этому конкретному заказу, так и указатель на другую запись заказа. Тогда в приложении-конверторе валют, которым мы занимались ранее, можно было бы использовать структуру, которая выглядела бы примерно так (рис. 1.1):

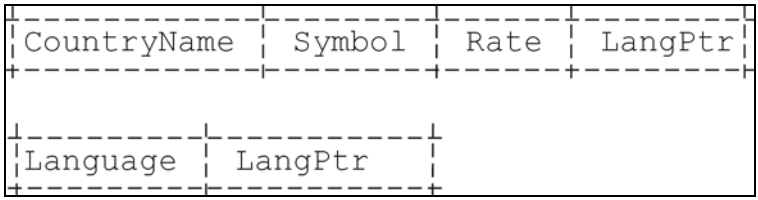


Рис. 1.1. Структура записей конвертора валют

Данные загружаются и получается связанный (отсюда и название модели – *сетевая*) список для языков (рис. 1.2):

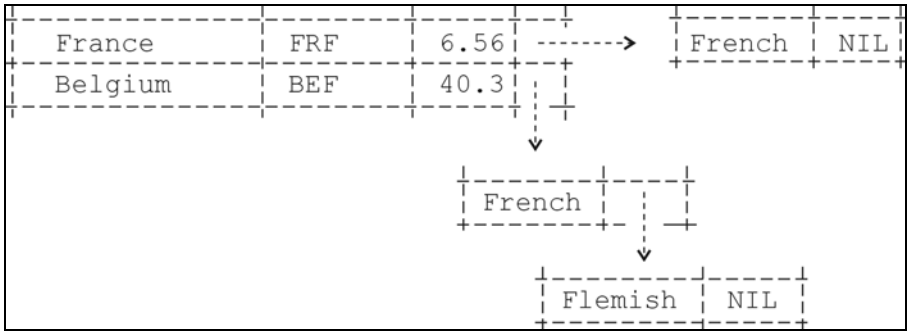


Рис. 1.2. Связанный список

Два разных типа записей, представленные на рисунке, будут храниться отдельно, каждый – в своей собственной таблице.

Конечно же, было бы более целесообразно, если бы названия языков не повторялись в базе снова и снова. Вероятно, лучше ввести третью таблицу, в которой содержались бы языки и идентификатор (часто в этом качестве используется целое число), который бы использовался для ссылки на запись таблицы языков из записей другого типа. Такой идентификатор называется ключом.

У сетевой модели базы данных есть несколько важных преимуществ. Если нужно найти все записи одного типа, относящиеся к определенной записи другого типа (например, языки, на которых говорят в одной из стран), то можно сделать это очень быстро, следуя по указателям, начиная с указанной записи.

Есть, однако, и недостатки. Если нам нужен перечень стран, в которых говорят по-французски, придется пройти по ссылкам всех записей стран, и для больших баз данных такая операция будет выполняться очень медленно. Это можно исправить, создав другие связанные списки указателей специально для языков, но такое решение быстро становится слишком сложным и, конечно же, не является универсальным, поскольку необходимо заранее решить, как будут организованы ссылки.

К тому же, писать приложение, использующее сетевую модель базы данных, достаточно утомительно, потому что обычно ответственность за создание и поддержание указателей по мере обновления и удаления записей лежит на приложении.

Иерархическая модель базы данных

В конце 1960-х годов IBM использовала в СУБД IMS иерархическую модель построения базы. В этой модели проблема повторяющихся групп решалась за счет представления одних записей как состоящих из множеств других.

Это можно представить как «спецификацию материалов», которая применяется для описания составляющих сложного продукта. Например, машина состоит (скажем) из шасси, кузова, двигателя и четырех колес. Каждый из этих основных компонентов в свою очередь состоит из некоторых других. Двигатель включает в себя несколько цилиндров, головку цилиндра и коленчатый вал. Эти компоненты опять-таки состоят из более мелких; так мы доходим до гаек и болтов, которыми комплектуются любые составляющие автомобиля.

Иерархическая модель базы данных применяется до сих пор. Иерархическая СУБД способна оптимизировать хранение данных в том, что касается некоторых отдельных вопросов, например можно без труда определить, в каком автомобиле используется какая-то конкретная деталь.

Реляционная модель базы данных

Огромный скачок в развитии теории систем управления базами данных произошел в 1970 году, когда был опубликован доклад Е. Ф. Кодда (E. F. Codd) «Реляционная модель для больших разделяемых банков данных» («A Relational Model of Data for Large Shared Data Banks»), см. *http://www.acm.org/classics/nov95/toc.html*. В этом поис-

тине революционном труде вводилось понятие отношений и было показано, как использовать таблицы для представления фактов, которые устанавливают отношения с объектами «реального мира» и, следовательно, хранят данные о них.

К этому времени уже стало очевидно, что эффективность, достижение которой первоначально являлось основополагающим при проектировании базы, не так важна, как целостность данных. Реляционная модель придает гораздо большее значение целостности данных, чем любая другая ранее применявшаяся модель.

Реляционную систему управления базами данных определяет набор правил. Во-первых, запись таблицы носит название «кортеж», и именно этот термин используется в части документации на PostgreSQL. Кортеж – это упорядоченная группа компонентов (или атрибутов), каждый из которых принадлежит определенному типу. Все кортежи построены по одному шаблону, во всех одинаковое количество компонентов одинаковых типов. Приведем пример набора кортежей:

```
{"France", "FRF", 6.56}
{"Belgium", "BEF", 40.1}
```

Каждый из этих кортежей состоит из трех атрибутов: названия страны (строковый тип), валюты (строковый тип) и валютного курса (тип с плавающей точкой). В реляционной базе данных все записи, добавляемые в это множество (или таблицу), должны следовать этой же форме, поэтому записи, представленные ниже, не могут быть добавлены:

```
{"Germany", "DEM"}
    - недостаточно атрибутов
{"Switzerland", "CHF", "French", "German", "Italian", "Romansch"}
     - слишком много атрибутов
{1936.27, "ITL", "Italy"}
    - неправильный тип атрибутов (неправильный порядок)
```

Более того, ни в одной таблице не может быть повторения кортежей. То есть в любой таблице реляционной базы данных повторяющиеся строки или записи не разрешены.

Такая мера может выглядеть как драконовская, поскольку может показаться, что для системы, которая хранит заказы, размещаемые клиентами, это означает, что один клиент не сможет заказать какой-то продукт дважды. В следующей главе будет показано, что на практике этот запрет легко обойти, введя дополнительный атрибут.

Каждый атрибут записи должен быть «атомарным», то есть представлять собой простую порцию информации, а не другую запись или список других аргументов. Кроме того, типы соответствующих атрибутов в каждой записи должны совпадать, как было показано выше. Технически это означает, что они должны быть получены из одного и того же набора значений или домена. Практически же все они должны быть

или строками, или целыми числами, или числами с плавающей точкой, или же принадлежать какому-то другому типу, поддерживаемому СУБД.

Атрибут, по которому отличают записи, во всем остальном идентичные, называется ключом. В некоторых случаях в качестве ключа может выступать комбинация из нескольких атрибутов.

Атрибут (или атрибуты), предназначенный для того, чтобы отличить некоторую запись таблицы от всех остальных записей этой таблицы (или, другими словами, сделать запись уникальной), называется первичным ключом. В реляционной базе данных каждое отношение (таблица) должно иметь первичный ключ, то есть что-то, что делало бы каждую запись отличной от всех остальных в этой таблице.

Последнее правило, определяющее структуру реляционной базы данных, – это **ссылочная целостность**. Такое требование объясняется тем, что в любой момент времени все записи базы данных должны иметь смысл. Разработчик приложения, взаимодействующего с базой данных, должен быть внимателен, он обязан убедиться, что его код не нарушает целостности базы. Представьте, что происходит при удалении клиента. Если клиент удаляется из отношения `CUSTOMER`, необходимо удалить и все его заказы из таблицы `ORDERS`. В противном случае останутся записи о заказах, которым не сопоставлен клиент.

В следующих главах будет представлена более подробная теоретическая и практическая информация о реляционных базах данных. Пока же запомните, что реляционная модель построена на таких математических понятиях, как множества и отношения, и что при создании систем следует придерживаться определенных правил.

Языки запросов

Реляционные системы управления базами данных, конечно же, предоставляют способы добавления и обновления данных, но это не главное, сила таких систем заключается в том, что они предоставляют пользователю возможность задавать вопросы о хранимых данных на специальном языке запросов. В отличие от более ранних баз данных, которые специально проектировались так, чтобы отвечать на определенные типы вопросов, касающихся содержащейся в них информации, реляционные базы данных являются гораздо более гибкими и отвечают на вопросы, которые еще не были известны при создании базы.

Предложенная Коддом реляционная модель использует тот факт, что отношения определяют множества, а множества можно обрабатывать математически. Кодд предположил, что в запросах мог бы применяться такой раздел теоретической логики, как исчисление предикатов, на его основе и построены языки запросов. Такой подход обеспечивает

беспрецедентную производительность поиска и выборки множеств данных.

Одним из первых был реализован язык запросов QUEL, он использовался в созданной в конце 1970х годов базе данных Ingres. Еще один язык запросов, в котором применялся другой метод, назывался QBE (Query By Example – запрос по примеру). Приблизительно в то же самое время группа, работающая в исследовательском центре IBM, разработала язык структурированных запросов SQL (Structured Query Language), это название обычно произносится как «сиквел».

SQL – это стандартный язык, наиболее распространенным его определением является стандарт **ISO/IEC 9075:1992, «Information Technology – Database Languages – SQL»** (или, проще говоря, SQL92) и его американский аналог ANSI X3.135-1992, отличающийся от первого лишь несколькими страницами обложки. Эти стандарты заменили ранее существовавший SQL89. На самом деле есть и более поздний стандарт, SQL99, но он еще не получил распространения, к тому же большая часть обновлений не затрагивает ядро языка SQL.

Существуют три уровня соответствия SQL92: Entry SQL, Intermediate SQL и Full SQL. Самым распространенным является уровень «Entry», и PostgreSQL очень близок к такому соответствию, хотя есть и небольшие различия. Разработчики занимаются исправлением незначительных упущений, и с каждой новой версией PostgreSQL становится все ближе к стандарту.

В языке SQL три типа команд:

- **Data Manipulation Language (DML) – язык манипулирования данными**

  Это та часть SQL, которая используется в 90% случаев. Она состоит из команд добавления, удаления, обновления и, что важнее всего, выборки данных из базы данных.

- **Data Definition Language (DDL) – язык определения данных**

  Это команды для создания таблиц и управления другими аспектами базы данных, структурированными на более высоком уровне, чем относящиеся к ним данные.

- **Data Control Language (DCL) – язык управления данными**

  Это набор команд, контролирующих права доступа к данным. Многие пользователи баз данных никогда не применяют такие команды, поскольку работают в больших компаниях, где есть специальный администратор базы данных (или даже несколько), который занимается управлением базой данных, в его функции входит и контроль за правами доступа.

С каждой страницей вы будете узнавать об SQL все больше и больше, так что, когда доберетесь до конца, не только познакомитесь с многими операторами этого языка, но и будете знать, как их использовать.

SQL

SQL практически повсеместно признан стандартным языком запросов и, как уже упоминалось, описан во многих международных стандартах. В наши дни почти каждая СУБД в той или иной степени поддерживает SQL. Это способствует унификации, т. к. приложение, написанное с применением SQL в качестве интерфейса к базе данных, может быть перенесено и использоваться на другой базе, при этом стоимость такого переноса в терминах затраченного времени и прилагаемых усилий будет невелика.

Однако под давлением рынка производители баз данных вынуждены создавать отличающиеся друг от друга продукты. Так появилось несколько диалектов SQL, чему способствовало и то, что в стандарте, описывающем язык, не определены команды для многих задач администрирования базы данных, которые представляют собой необходимую и очень важную составляющую при использовании базы в реальном мире. Поэтому существуют различия между диалектами SQL, принятыми (например) в Oracle, SQL Server и PostgreSQL.

SQL будет описываться на протяжении всей книги, пока же приведем несколько примеров, чтобы показать, на что этот язык похож. Оказывается, для того чтобы начать работать с SQL, не обязательно изучать его формальные правила.

Создадим при помощи SQL новую таблицу в базе данных. В этом примере создается таблица для товаров, предлагаемых на продажу, которые войдут в заказ:

```
CREATE TABLE item
(
    item_id                 serial,
    description             char(64)          not null,
    cost_price              numeric(7,2),
    sell_price              numeric(7,2)
);
```

Здесь мы определили, что таблице необходим идентификатор, который бы действовал как первичный ключ, и что он должен автоматически генерироваться системой управления базой данных. Идентификатор имеет тип `serial`, а это означает, что каждый раз при добавлении нового элемента `item` в последовательности будет создан новый, уникальный `item_id`. Описание (`description`) – это текстовый атрибут, состоящий из 64 символов. Себестоимость (`cost_price`) и цена продажи (`sell_price`) определяются как числа с плавающей точкой, с двумя знаками после запятой.

Теперь используем SQL для заполнения только что созданной таблицы. В этом нет ничего сложного:

```
INSERT INTO item(description, cost_price, sell_price)
values('Fan Small', 9.23, 15.75);
```

```
INSERT INTO item(description, cost_price, sell_price)
values('Fan Large', 13.36, 19.95);
INSERT INTO item(description, cost_price, sell_price)
values('Toothbrush', 0.75, 1.45);
```

Основа SQL – это оператор SELECT. Он применяется для создания результирующих множеств – групп записей (или атрибутов записей), которые соответствуют некоторому критерию. Эти критерии могут быть достаточно сложными. Результирующие множества могут использоваться в качестве целевых объектов для изменений, осуществляемых оператором UPDATE, или удалений, выполняемых DELETE.

Вот несколько примеров использования оператора SELECT:

```
SELECT * FROM customer, orderinfo
WHERE orderinfo.customer_id = customer.customer_id GROUP BY customer_id
```

```
SELECT customer.title, customer.fname, customer.lname,
COUNT(orderinfo.orderinfo_id) AS "Number of orders" FROM customer, orderinfo
WHERE customer.customer_id = orderinfo.customer_id
GROUP BY customer.title, customer.fname, customer.lname
```

Эти операторы SELECT перечисляют все заказы клиентов в указанном порядке и подсчитывают количество заказов, сделанных каждым клиентом. Результаты работы данных операторов приведены в главе 2, а подробно о SELECT будет рассказано в главе 4.

PostgreSQL предоставляет несколько способов доступа к данным, в частности можно:

- Использовать консольное приложение для выполнения операторов SQL
- Непосредственно встроить SQL в приложение
- Использовать вызовы функций API (Application Programming Interfaces, интерфейсов прикладного программирования) для подготовки и выполнения операторов SQL, просмотра результирующих множеств и обновления данных из множества различных языков программирования
- Прибегнуть к опосредованному доступу к данным базы PostgreSQL с применением драйвера ODBC (Open Database Connection – открытого интерфейса доступа к базам данных) или JDBC (Java Database Connectivity – интерфейса доступа Java-приложений к базам данных) или стандартной библиотеки, такой как DBI для языка Perl

Системы управления базами данных

СУБД, как уже говорилось ранее, – это набор программ, делающих возможным построение баз данных и их использование. В обязанности СУБД входит:

- **Создание базы данных**

  Некоторые системы управляют одним большим файлом и создают одну или несколько баз данных внутри него, другие могут задействовать несколько файлов операционной системы или же непосредственно реализовывать низкоуровневый доступ к разделам диска. Пользователи и разработчики не должны заботиться о низкоуровневой структуре таких файлов, т. к. весь необходимый доступ обеспечивает СУБД.

- **Предоставление средств для выполнения запросов и обновлений**

  СУБД должна обеспечивать возможность запроса данных, удовлетворяющих некоторому критерию, например возможность выбора всех заказов, сделанных некоторым клиентом, но еще не доставленных. До того как SQL получил широкое распространение в качестве стандартного языка, способы выражения таких запросов менялись от системы к системе.

- **Многозадачность**

  Если с базой данных работают несколько приложений или к ней одновременно осуществляют доступ несколько пользователей, то СУБД должна гарантировать, что обработка запроса каждого пользователя не влияет на работу остальных. То есть пользователям приходится ждать, только если кто-то другой записывает данные именно тогда, когда им нужно прочитать (или записать) данные в какой-то элемент. Одновременно может происходить несколько считываний данных. На поверку оказывается, что разные базы данных поддерживают разные уровни многозадачности и что эти уровни даже могут быть настраиваемыми. Мы поговорим об этом в главе 9.

- **Ведение журнала**

  СУБД должна вести журнал всех изменений данных за некоторый период времени. Он может использоваться для отслеживания ошибок, а также (может быть, это даже важнее) для восстановления данных в случае сбоя системы, например внепланового выключения питания. Обычно производится резервное копирование данных и ведется журнал транзакций, т. к. резервная копия может быть полезна для восстановления базы данных в случае повреждения диска.

- **Обеспечение безопасности базы данных**

  СУБД должна обеспечивать контроль над доступом, чтобы только зарегистрированные пользователи могли манипулировать данными, хранящимися в базе, и самой структурой базы данных (атрибутами, таблицами и индексами). Обычно для каждой базы определяется иерархия пользователей, во главе этой структуры стоит «суперпользователь», который может изменять все что угодно, дальше идут пользователи, которые могут добавлять и удалять дан-

ные, а в самом низу находятся те, кто имеет право только на чтение. СУБД должна иметь средства, позволяющие добавлять и удалять пользователей, а также указывать, к каким возможностям базы данных они могут получить доступ.

- **Поддержание ссылочной целостности**

  Многие СУБД имеют свойства, способствующие поддержанию ссылочной целостности, то есть корректности данных. Обычно, если запрос или обновление нарушает правила реляционной модели, СУБД выдает сообщение об ошибке.

Что такое PostgreSQL?

Пришло время рассказать о том, что же такое PostgreSQL. Это СУБД, которая использует реляционную модель для своих баз данных и поддерживает стандартный язык запросов SQL.

PostgreSQL предоставляет множество различных возможностей, достаточно надежна и имеет хорошие характеристики по производительности. Она работает практически на всех UNIX-платформах, включая UNIX-подобные системы, такие как FreeBSD и Linux. Ее можно применять на Windows NT Server и Windows 2000 Server, а для разработки годятся даже такие системы Microsoft для рабочих станций, как ME. Кроме того, PostgreSQL свободно распространяется и имеет открытый исходный код.

PostgreSQL выгодно отличается от многих других СУБД. Она обладает практически всеми возможностями, которые есть в других базах данных (коммерческих или Open Source), а также некоторыми дополнительными.

Приведем перечень функциональных возможностей PostgreSQL (представлен в ответах на часто задаваемые вопросы по PostgreSQL):

- Транзакции
- Вложенные запросы
- Представления
- Ссылочная целостность – внешние ключи
- Сложные блокировки
- Типы, определяемые пользователем
- Наследственность
- Правила
- Проверка совместимости версий

Обо всем этом и рассказывается в данной книге.

Начиная с версии 6.5 PostgreSQL представляет собой весьма устойчивую систему, каждая следующая версия проходит процедуру регрес-

сивного тестирования, обеспечивающего стабильность. Версия 7.x PostgreSQL как никогда близка к соответствию стандарту SQL92, устранено раздражавшее ограничение размера строки.

В ходе эксплуатации PostgreSQL проявила себя как заслуживающая доверия СУБД. Каждая версия проверяется очень тщательно, бета-версии проходят как минимум месячное тестирование. Благодаря многочисленному сообществу пользователей и открытому доступу к исходному коду ошибки исправляются очень быстро.

Производительность этой СУБД также возрастает от версии к версии, и последние аттестации показывают, что при определенных условиях она не уступает коммерческим продуктам. Некоторые системы, обладающие не таким полным набором возможностей, превосходят PostgreSQL в производительности, но за счет потери функциональности. Для достаточно простых приложений такую роль играет база данных, состоящая из плоских файлов.

Короткий экскурс в историю PostgreSQL

Генеалогическое древо PostgreSQL начинается в 1977 году в Калифорнийском университете Беркли. Реляционная база данных Ingres разрабатывалась в Беркли в 1977–85 годах. Ingres была очень популярна за пределами стен Калифорнийского университета, она появилась на многих компьютерах в университетских и исследовательских кругах. На свободный рынок Ingres была выведена Relational Technologies/Ingres Corporation, так она стала одной из первых коммерчески доступных реляционных систем управления базами данных. В наши дни Ingres превратилась в CA-INGRES II, продукт Computer Associates.

Тем временем в Беркли не прекращается работа над сервером реляционной базы данных (под названием Postgres), это продолжается с 1986 по 1994 год. Как и раньше, код приобретается коммерческой компанией и продукт на его основе выставляется на продажу. После поглощения компанией Informix он стал называться Illustra. Где-то в 1994 году в Postgres были добавлены возможности SQL, и возникло новое имя – Postgres95.

К 1996 году Postgres стала приобретать необыкновенную популярность, и было принято решение открыть ее код для некоторого количества программистов по списку рассылки, так началось весьма успешное сотрудничество добровольцев, направленное на продвижение Postgres. Тогда продукт в последний раз сменил имя, отбросив окончание «95» и заменив его на «SQL», которое лучше отражало поддержку стандартного языка запросов SQL в Postgres. Так родилась PostgreSQL.

Сейчас PostgreSQL развивается группой интернет-разработчиков, приблизительно тем же способом, что и другое программное обеспечение Open Source: Perl, Apache и PHP. Пользователи имеют доступ к ис-

ходным кодам, они исправляют ошибки, занимаются совершенствованием продукта, предлагают введение новых возможностей. Официальные версии PostgreSQL выпускаются на *http://www.postgresql.org*.

GreatBridge осуществляет коммерческую поддержку проекта, а также предоставляет работу некоторым PostgreSQL-разработчикам. (Более подробная информация приведена в разделе «Ресурсы» в конце главы.)

Архитектура PostgreSQL

Одной из сильных сторон PostgreSQL является ее архитектура. Как и многие коммерческие СУБД, PostgreSQL может применяться в среде клиент-сервер, что дает массу преимуществ как пользователям, так и разработчикам.

Основа PostgreSQL составляет серверный процесс базы данных. Он выполняется на одном сервере. (В этой СУБД еще не реализована технология высокой готовности, как в некоторых других коммерческих системах уровня предприятия, которые могут распределять нагрузку между несколькими серверами, добиваясь таким образом дополнительной масштабируемости и устойчивости к внешним воздействиям.)

Доступ из приложений к данным базы осуществляется посредством процесса базы данных. Клиентские программы не могут получить доступ к данным самостоятельно, даже если они работают на том же компьютере, на котором выполняется серверный процесс.

Такое разделение клиентов и сервера позволяет построить распределенную систему. Можно отделить клиентов от сервера посредством сети и разрабатывать клиентские приложения в среде, удобной для пользователя. Например, можно реализовать базу данных под UNIX и создать клиентские приложения, которые будут работать в системе Microsoft Windows.

Приведенная ниже схема (рис. 1.3) показывает типичную модель распределенного приложения PostgreSQL:

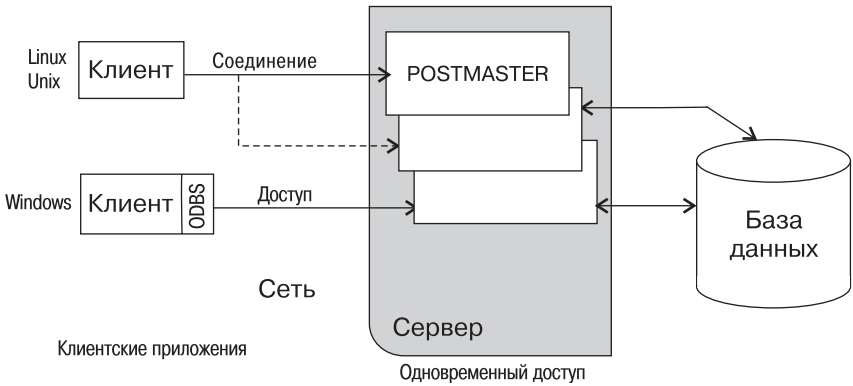


Рис. 1.3. Работа типичного приложения PostgreSQL

Несколько клиентов подсоединяются к серверу по сети. PostgreSQL ориентирован на протокол TCP/IP – это может быть локальная сеть или Интернет. Каждый клиент соединяется с основным серверным процессом базы данных (на схеме – Postmaster), который создает новый серверный процесс специально для обслуживания запросов на доступ к данным конкретного клиента.

Благодаря тому что манипулирование данными сосредоточено на сервере, СУБД не приходится контролировать многочисленных клиентов, получающих доступ в совместно используемый каталог сервера, и PostgreSQL может поддерживать целостность данных даже при одновременном доступе большого количества пользователей.

Клиентские приложения соединяются с базой по специальному протоколу PostgreSQL. Однако можно установить на стороне клиента программное обеспечение, предоставляющее стандартный интерфейс для работы с нужным приложением, например по стандарту ODBC или JDBC. Доступность ODBC-драйвера позволяет применять PostgreSQL в качестве базы данных для многих существующих приложений, включая такие продукты Microsoft Office, как Excel и Access. Способность взаимодействия PostgreSQL с приложениями будет подробнее рассмотрена в следующих главах.

Архитектура клиент-сервер делает возможным разделение труда. Машина-сервер хорошо подходит для хранения и управления доступом к большим объемам данных, она может использоваться как надежный репозитарий. Для клиентов могут быть разработаны сложные графические приложения. В качестве альтернативы можно создать внешний интерфейс на основе Интернета, который предоставлял бы доступ к данным и возвращал результат в виде веб-страниц в стандартный веб-броузер, при этом не требовалось бы никакого дополнительного клиентского программного обеспечения. Вернемся и к этой идее в следующей главе.

Open Source–лицензирование

В начале XXI века многие компьютерные системы создаются на основе свободно распространяемых программ с открытыми исходными кодами. К их числу относится и PostgreSQL. Что же это означает в действительности?

Когда понятие Open Source применяется к программному обеспечению, оно приобретает специальный смысл. Такое программное обеспечение поставляется вместе с исходными кодами. Это не обязательно значит, что на его применение не налагаются никакие условия. Оно все-таки лицензируется в том смысле, что вы получаете право некоторым образом использовать это программное обеспечение.

Open Source-лицензия дает право на использование, модификацию и распространение программного обеспечения без лицензионных вы-

плат. То есть вы можете работать с PostgreSQL в своей компании так, как это удобно в вашем случае.

Если с программным обеспечением Open Source возникают проблемы, пользователь может или исправить ошибки сам (поскольку у него есть исходные тексты), или же передать код кому-то другому для исправления. Сейчас многие коммерческие компании предлагают поддержку продуктов Open Source, поэтому, приобретая такой продукт, не стоит чувствовать себя «забытым».

Существует несколько разновидностей лицензий Open Source, некоторые из них имеют большее число ограничений на распространение, другие меньшее. Тем не менее все они придерживаются принципа доступности исходных кодов программ и разрешают дальнейшее их распространение.

Наиболее либеральной является лицензия Berkeley Software Distribution (BSD), разрешающая делать с программным обеспечением все что угодно, не предоставляя при этом никаких гарантий. Лицензия на использование PostgreSQL по сути своей аналогична лицензии BSD, она представляет собой заявление об авторских правах, в котором говорится: «Настоящим предоставляется право на использование, копирование, модификацию и распространение данного программного продукта и относящейся к нему документации в любых целях, без оплаты и без подписания соответствующих соглашений при условии, что во всех копиях будет присутствовать уведомление об авторских правах, указанное выше, данный абзац и два последующих». Два следующих абзаца посвящены отказу от каких бы то ни было обязательств и гарантий.

Ресурсы

Информация о базах данных в целом и о PostgreSQL в частности может быть получена из множества источников, как печатных, так и доступных через Интернет.

Подробно о теоретических основах баз данных рассказано в разделе Tech Talk на сайте Дэвида Фрика (David Frick) *http://www.frick-cpa.com*.

Официальным сайтом PostgreSQL является *http://www.postgresql.org*, там можно ознакомиться с историей СУБД, скачать копию PostgreSQL, просмотреть официальную документацию, а также сделать много других вещей, например научиться правильно произносить название «PostgreSQL».

Коммерческая поддержка PostgreSQL предоставляется Great Bridge на сайте *http://www.greatbridge.com*. Great Bridge также поддерживает сайт Open Source проекта *http://www.greatbridge.org*, где можно ознакомиться с последними новинками, относящимися к PostgreSQL,

такими как инструменты для администрирования, графические интерфейсы пользователя и т. д.

Red Hat Database также поддерживает проект PostgreSQL. Материалы о Red Hat Database можно найти по адресу: *http://www.redhat.com/products/software/database/*.

Некоторые полезные вещи для PostgreSQL можно найти на сайте PostgreSQL Inc. *http://www.pgsql.com*.

Тем, кого интересует тема свободного распространения и открытости исходных кодов в отношении программного обеспечения (Open Source продукты), советуем посетить два следующих сайта:

- *http://www.gnu.org*
- *http://www.opensource.org*

2

Основы реляционных баз данных

В этой главе рассказывается о том, что делает СУБД и, в частности, реляционные СУБД, такие как PostgreSQL, столь полезными для хранения данных. Начнем с рассмотрения электронных таблиц, у которых много общего с реляционными базами данных, но есть и существенные ограничения.

Будут продемонстрированы преимущества реляционных СУБД (на примере PostgreSQL) по сравнению с электронными таблицами и показано, как даже в очень простой базе данных извлечь выгоду из возможностей PostgreSQL. По ходу дела будем продолжать неформальное знакомство с SQL.

Итак, основные темы, рассматриваемые в данной главе:

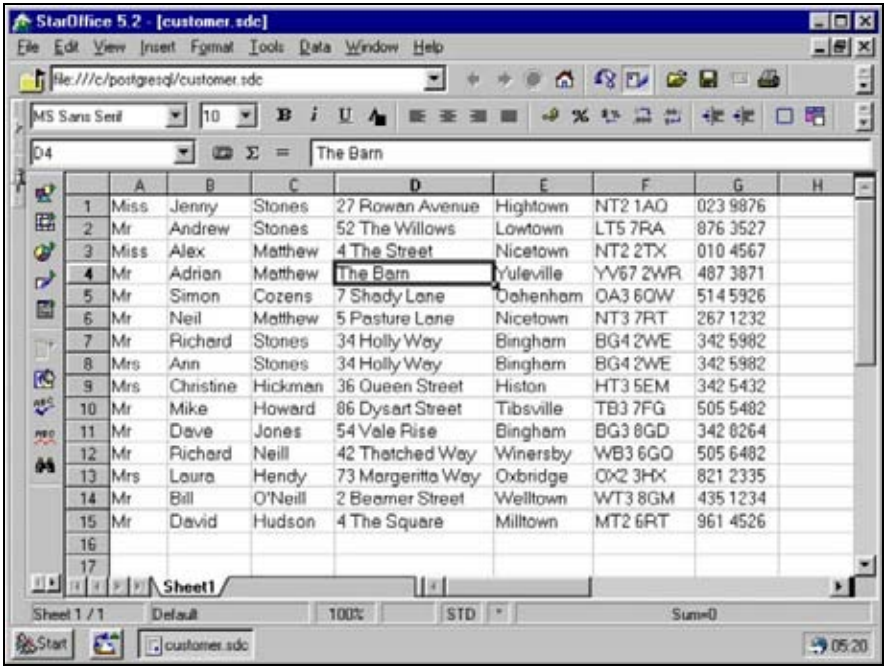
- Электронные таблицы – проблемы и ограничения
- Отличие таблицы от базы данных
- Данные в базе данных
- Основы проектирования базы данных с несколькими таблицами
- Установление отношений между таблицами
- Создание собственных таблиц – несколько практических приемов
- Демонстрация базы данных Клиенты/Заказы
- Некоторые стандартные типы данных и `NULL`

Электронные таблицы

Приложения, работающие с электронными таблицами, такие как Microsoft Excel, широко применяются для хранения и исследования данных. В самом деле, электронная таблица – это удобный способ размещения данных и просмотра их различными способами. Можно без труда отсортировать данные по столбцам и тогда, чтобы узнать о каких-то свойствах данных, достаточно просто взглянуть на них, если, конечно, их объем не слишком велик.

К сожалению, инструмент, удобный для просмотра и манипулирования данными, часто считают подходящим для хранения и совместного использования сложных данных. Но во многих случаях это не так.

Большинство пользователей хорошо знакомы с одной или несколькими электронными таблицами и свободно обращаются с данными, размещенными в несколько строк и столбцов, например представленными в электронной таблице StarOffice, которая хранит список клиентов (рис. 2.1):



	A	B	C	D	E	F	G	H
1	Miss	Jenny	Stones	27 Rowan Avenue	Hightown	NT2 1AQ	023 9876	
2	Mr	Andrew	Stones	52 The Willows	Lowtown	LT5 7RA	876 3527	
3	Miss	Alex	Matthew	4 The Street	Nicetown	NT2 2TX	010 4567	
4	Mr	Adrian	Matthew	The Barn	Yuleville	YV67 2WR	487 3871	
5	Mr	Simon	Cozens	7 Shady Lane	Oahenham	OA3 6QW	514 5926	
6	Mr	Neil	Matthew	5 Pasture Lane	Nicetown	NT3 7RT	267 1232	
7	Mr	Richard	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982	
8	Mrs	Ann	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982	
9	Mrs	Christine	Hickman	36 Queen Street	Histon	HT3 5EM	342 5432	
10	Mr	Mike	Howard	86 Dysart Street	Tibsville	TB3 7FG	505 5482	
11	Mr	Dave	Jones	54 Vale Rise	Bingham	BG3 8GD	342 8264	
12	Mr	Richard	Neill	42 Thatched Way	Winersby	WB3 6GQ	505 6482	
13	Mrs	Laura	Hendy	73 Margeritta Way	Oxbridge	OX2 3HX	821 2335	
14	Mr	Bill	O'Neill	2 Beamer Street	Welltown	WT3 8GM	435 1234	
15	Mr	David	Hudson	4 The Square	Milltown	MT2 6RT	961 4526	
16								
17								

Рис. 2.1. Электронная таблица StarOffice с информацией о клиентах

Безусловно, такую информацию легко просматривать и модифицировать. Вероятно, даже не задумываясь об этом, мы сконструировали эту таблицу так, что она обладает некоторыми свойствами, которые окажутся удобными при проектировании баз данных. Каждому клиенту

отведена отдельная строка, а каждый фрагмент информации о клиенте хранится в отдельном столбце. Например, имя и фамилия хранятся в разных столбцах, так что если потребуется выполнить сортировку по фамилии, это не создаст проблем.

Немного терминологии

Прежде чем двигаться дальше, определим несколько терминов на примере небольшого участка данных. Наверняка вы все знаете о строках и столбцах, но давайте освежим эти знания (рис. 2.2):

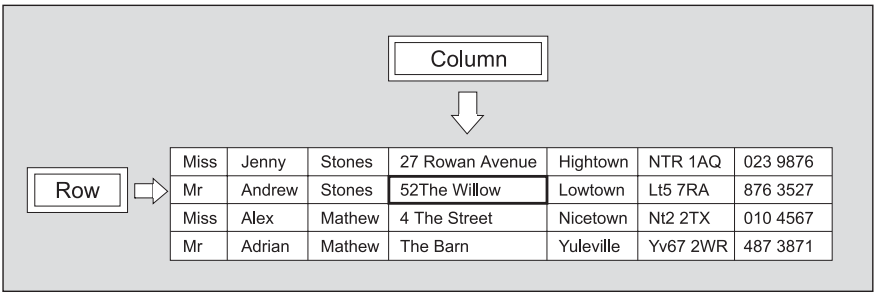


Рис. 2.2. Терминология электронных таблиц

Пересечение строки со столбцом образует ячейку. На снимке экрана StarOffice, представленном выше, ячейка с адресом «52 The Willow» выделена с помощью стрелок.

Недостатки электронных таблиц

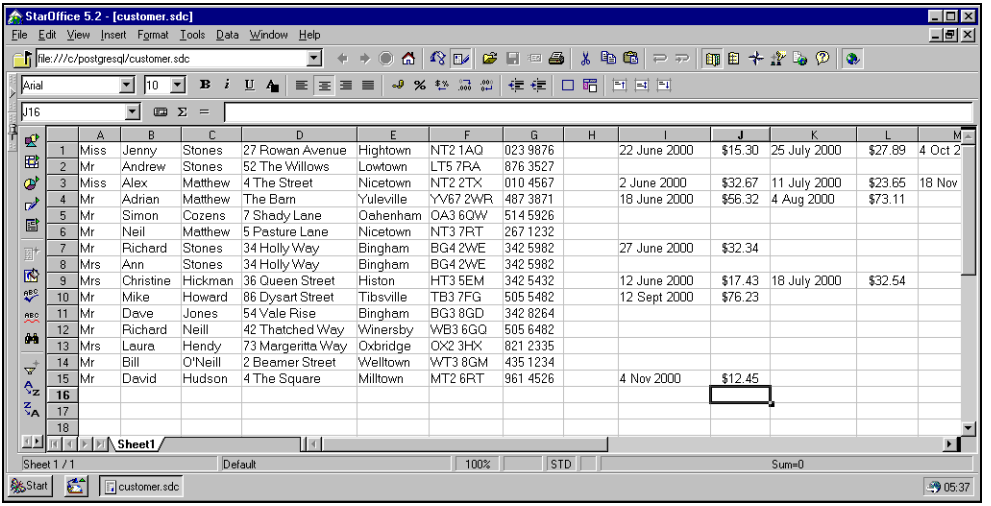
Чем же плохо хранить информацию о клиентах в электронной таблице? Может быть, и ничем, если клиентов не слишком много, как и информации о каждом из них. А также если не нужно размещать данные, например о заказах, которые эти клиенты сделали, и если не очень много пользователей хотят одновременно обновлять информацию. Список возможных проблем зависит от конкретных обстоятельств.

Электронные таблицы очень удобны во многих и многих случаях, это замечательный инструмент для решения ряда задач. Однако, подобно тому как вы не стали бы (по крайней мере не следовало бы этого делать) забивать гвозди отверткой, в некоторых ситуациях не стоит использовать электронные таблицы.

Просто представьте себе, что большая компания с тысячами клиентов хранит их список в простой таблице. В большой компании обновления в такой список вносят, скорее всего, несколько человек. Блокировка файла может гарантировать, что в каждый момент времени только один пользователь обновляет список, при этом количество служащих,

пытающихся ввести обновления, растет, и ждать своей очереди на редактирование им приходится все дольше и дольше. Хотелось бы сделать так, чтобы несколько человек могли одновременно читать, корректировать, добавлять и удалять строки. Очевидно, что простой блокировки недостаточно для того, чтобы эффективно решить эту проблему.

Предположим, что также необходимо хранить подробную информацию о каждом заказе клиента. Если размещать эти данные следом за сведениями о каждом заказчике, то при увеличении количества заказов таблица становится все более и более сложной. Посмотрите, что получится, если попытаться вводить некоторую базовую информацию о заказах рядом с данными их заказчика (рис. 2.3):



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Miss	Jenny	Stones	27 Rowan Avenue	Hightown	NT2 1AQ	023 9876		22 June 2000	$15.30	25 July 2000	$27.89	4 Oct 2
2	Mr	Andrew	Stones	52 The Willows	Lowtown	LT5 7RA	876 3527						
3	Miss	Alex	Matthew	4 The Street	Nicetown	NT2 2TX	010 4567		2 June 2000	$32.67	11 July 2000	$23.65	18 Nov
4	Mr	Adrian	Matthew	The Barn	Yuleville	YV67 2WR	487 3871		18 June 2000	$56.32	4 Aug 2000	$73.11	
5	Mr	Simon	Cozens	7 Shady Lane	Oahenham	OA3 6QW	514 5926						
6	Mr	Neil	Matthew	5 Pasture Lane	Nicetown	NT3 7RT	267 1232						
7	Mr	Richard	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982		27 June 2000	$32.34			
8	Mrs	Ann	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982						
9	Mrs	Christine	Hickman	36 Queen Street	Histon	HT3 5EM	342 5432		12 June 2000	$17.43	18 July 2000	$32.54	
10	Mr	Mike	Howard	86 Dysart Street	Tibsville	TB3 7FG	505 5482		12 Sept 2000	$76.23			
11	Mr	Dave	Jones	54 Vale Rise	Bingham	BG3 8GD	342 8264						
12	Mr	Richard	Neill	42 Thatched Way	Winersby	WB3 6GQ	505 6482						
13	Mrs	Laura	Hendy	73 Margeritta Way	Oxbridge	OX2 3HX	821 2335						
14	Mr	Bill	O'Neill	2 Beamer Street	Welltown	WT3 8GM	435 1234						
15	Mr	David	Hudson	4 The Square	Milltown	MT2 6RT	961 4526		4 Nov 2000	$12.45			
16													
17													
18													

Рис. 2.3. Таблица с информацией о заказах

К сожалению, таблица уже не выглядит так красиво, как раньше. Теперь ее строки имеют произвольную длину, поэтому подсчитать, сколько клиент потратил денег, не так уж просто. Со временем будет превышено разрешенное для каждой строки количество столбцов. Это та же проблема повторяющихся групп, которая обсуждалась в предыдущей главе.

Покажем на примере, насколько легко выйти за пределы возможностей электронной таблицы. Некто пытался создать таблицу для друга, у которого было свое небольшое дело. Он занимался производством изделий из кожи, при этом цена изделия зависела не только от количества времени, затраченного на его создание, но и от цены на кожу, пошедшую на изготовление. Владелец покупал кожу партиями, при этом в каждой партии цена единицы материала отличалась от предыдущей. Запасы кожи расходовались в порядке поступления (первый

прибыл – первый использован), обычно из партии получалось несколько изделий. Требовалось создать электронную таблицу для:

- Наблюдения за складом
- Наблюдения за тем, сколько осталось партий кожи
- Отслеживания стоимости партии, использующейся в настоящий момент
- И, чтобы усложнить задачу, хранения количества различных сортов кожи

После нескольких дней попыток они поняли, что эта на первый взгляд простая задача отслеживания состояния имеющихся запасов неожиданно оказалась слишком сложной для решения при помощи электронной таблицы. Все дело в том, что количество записей должно быть переменным, что не очень-то согласуется с природой электронных таблиц.

Мы хотели показать, что электронные таблицы – это выдающееся средство для решения определенных задач, но область их применения ограничена.

В чем отличие таблицы от базы данных?

На первый взгляд кажется, что реляционная база данных, такая как PostgreSQL, во многом похожа на электронную таблицу, только является гораздо более гибким решением. Базы данных могут хранить значительно более сложную информацию и обладают рядом других свойств, например обеспечивают одновременный доступ нескольких пользователей, благодаря чему они удобнее для хранения данных.

Начнем с того, что сохраним в базе данных простой список клиентов, состоящий из одного листа, и посмотрим, какую выгоду можно извлечь из такого способа хранения. Позже в этой же главе будет рассказано о том, как PostgreSQL решает описанную ранее проблему с заказами клиентов.

Уже говорилось, что базы данных состоят из таблиц или, если применять формальную терминологию, из **отношений**. В этой книге авторы придерживаются понятия «таблица». Для начала необходимо спроектировать таблицу, в которой будет храниться информация о клиентах. Хорошая новость: уже имеющаяся электронная таблица является практически готовым решением, поскольку она хранит данные в некотором количестве строк и столбцов. Чтобы приступить к созданию основной таблицы базы данных, нужно ответить на три вопроса:

- Сколько нужно столбцов для хранения атрибутов, относящихся к каждому элементу?

- Какой тип данных будет храниться в каждом атрибуте (столбце)?
- Как различать разные строки, содержащие разные элементы?

Выбор столбцов

Вернемся к первоначальной электронной таблице с данными о клиентах. В ней уже представлен разумный набор столбцов для каждого клиента: имя, фамилия, почтовый индекс и т. д. С первым вопросом разобрались.

Выбор типа данных для каждого столбца

Теперь необходимо определить, данные какого типа будут находиться в каждом столбце. В электронной таблице каждая ячейка может иметь свой тип, а в базе данных данные в столбце должны иметь один и тот же тип. Как в большинстве языков программирования, столбцы базы данных принадлежат какому-либо типу. Чаще всего используются стандартные типы данных, поэтому обычно выбирать приходится между целыми числами, числами с плавающей точкой, текстом фиксированной длины, текстом переменной длины и датами. О типах данных PostgreSQL будет более подробно рассказано далее в этой главе, а также в главе 8. Часто легче всего выбрать соответствующий тип, просто посмотрев на примеры данных.

В нашем случае для всех столбцов годится текстовый тип, несмотря на то что номера телефонов представлены цифрами. Если хранить номер телефона как число, это может привести к потере начальный нулей, нельзя будет указывать международные коды (+), использовать скобки для кодов регионов и т. д. Не требуется особых усилий для того, чтобы понять, что во многих случаях номер телефона – это нечто большее, чем просто строка цифр.

Может быть, символьная строка тоже не является лучшим выбором для хранения телефонного номера (случайно можно сохранить посторонние символы), но в качестве отправной точки, несомненно, подходит больше, чем числовой тип. В дальнейшем всегда можно уточнить начальный проект. Очевидно, что длина слов первого столбца (Mr, Mrs, Dr) невелика и не превышает четырех символов. Аналогично, почтовые индексы также имеют ограниченную максимальную длину. Поэтому поля этих двух столбцов будут иметь фиксированную длину, а для всех остальных столбцов оставим длину переменной, потому что невозможно предугадать размер, например, фамилии человека.

Еще одно важное отличие базы данных от электронной таблицы заключается в том, что количество столбцов в таблице базы данных должно быть одинаковым во всех строках. В первой версии нашей электронной таблицы это требование выполняется.

Определение уникальности строк

Последнее изменение, которое необходимо внести, чтобы преобразовать электронную таблицу в базу данных, не столь очевидно, его необходимость вызвана особенностями управления отношениями между таблицами, существующими в базах данных. Следует решить, как сделать каждую строку с данными заказчика отличной от всех остальных клиентских строк в базе данных.

Другими словами, как различать клиентов? В электронной таблице не надо об этом заботиться, а вот при проектировании базы данных этот вопрос является ключевым, т. к. правила построения реляционных баз данных требуют, чтобы каждая строка была в некотором роде уникальной.

Казалось бы, на этот вопрос есть очевидный ответ – «по фамилии», но, к сожалению, такой вариант не проходит. Вполне возможно, что два клиента будут носить одну и ту же фамилию. Можно было бы выбрать в качестве отличительного признака номер телефона, но это не годится в том случае, если два клиента живут по одному и тому же адресу. Теперь можно дать волю воображению и предложить использовать комбинацию имени и номера телефона.

Конечно же, маловероятно, что два клиента с одинаковыми фамилиями могут иметь одинаковые номера телефонов, но (не говоря уже о том, что это решение не очень красиво) возникает проблема с сопровождением. Что если клиент сменит телефонную компанию и номер телефона? По нашему определению уникальности получается, что он должен стать новым клиентом, т. к. формально он отличается от того клиента, который существовал ранее, хотя на самом деле, конечно же, известно, что это один и тот же клиент с разными номерами телефонов.

При проектировании базы данных часто возникает вопрос об однозначной идентификации. Нужен «первичный ключ» – простой способ отличать одну строку с данными заказчика от остальных строк. К сожалению, цель еще не достигнута, но не все потеряно, стандартным решением является присваивание каждому клиенту уникального номера.

Все клиенты по очереди получают уникальный номер; так мы получаем возможность однозначной идентификации клиентов, на которую не влияют изменение номера телефона, переезд клиента и даже смена его фамилии. Проблема уникальности записей настолько распространена, что в PostgreSQL существует даже специальный тип данных, `serial`, используемый при ее решении. Далее в этой главе будет рассказано о таком типе данных.

Порядок строк

Есть еще одно важное отличие между данными, хранящимися в электронной таблице, и теми же данными в базе данных – в таблице поря-

док строк обычно бывает очень важен, в то время как в базе данных никакого порядка нет. Когда вы обращаетесь к таблице базы данных, строки могут быть выведены в любом порядке, если только нет явной инструкции упорядочить их по какому-либо признаку.

Размещение информации в базе данных

Теперь, когда структура базы данных выбрана, пришло время сохранить наши данные в базе. Позже мы еще вернемся к механизму определения таблицы базы данных, хранению и доступу к данным. На рис. 2.4 представлены данные таблицы PostgreSQL (для просмотра применялась простая утилита командной строки `psql` на машине с операционной системой Linux):

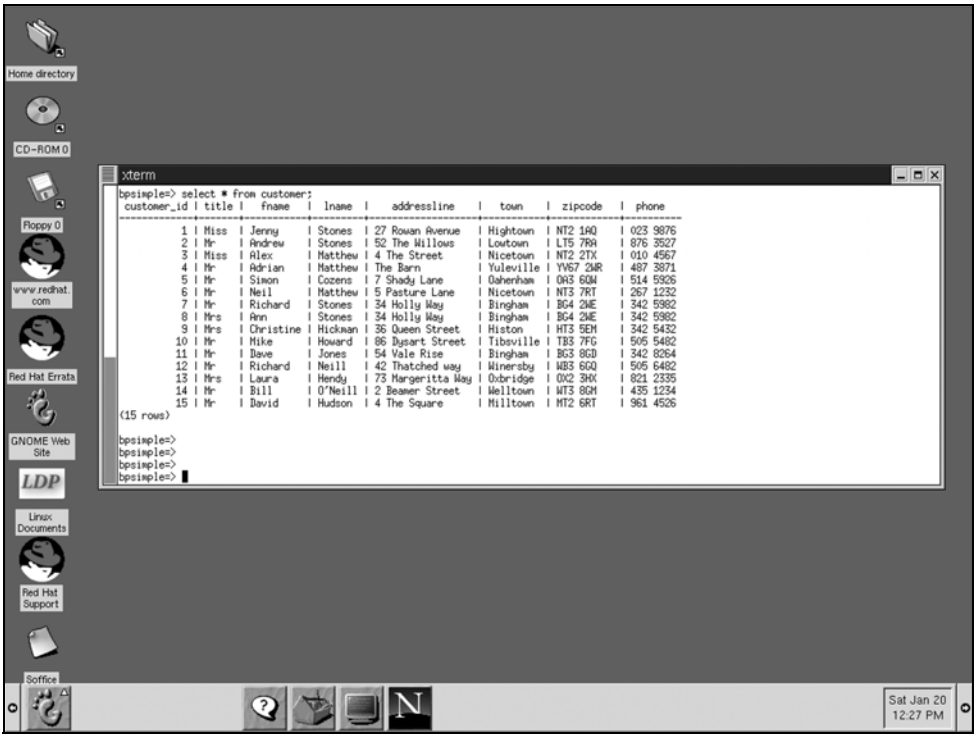


Рис. 2.4. Использование psql для просмотра таблицы PostgreSQL

Обратите внимание, что добавлен дополнительный столбец, `customer_id`, обеспечивающий однозначность ссылок на клиента. Он является первичным ключом для этой таблицы. Данные разбиты на строки и столбцы и напоминают электронную таблицу.

Обращаться к таблицам PostgreSQL можно не только из командной строки. Покажем ту же самую таблицу в графической программе pgAccess (рис. 2.5):

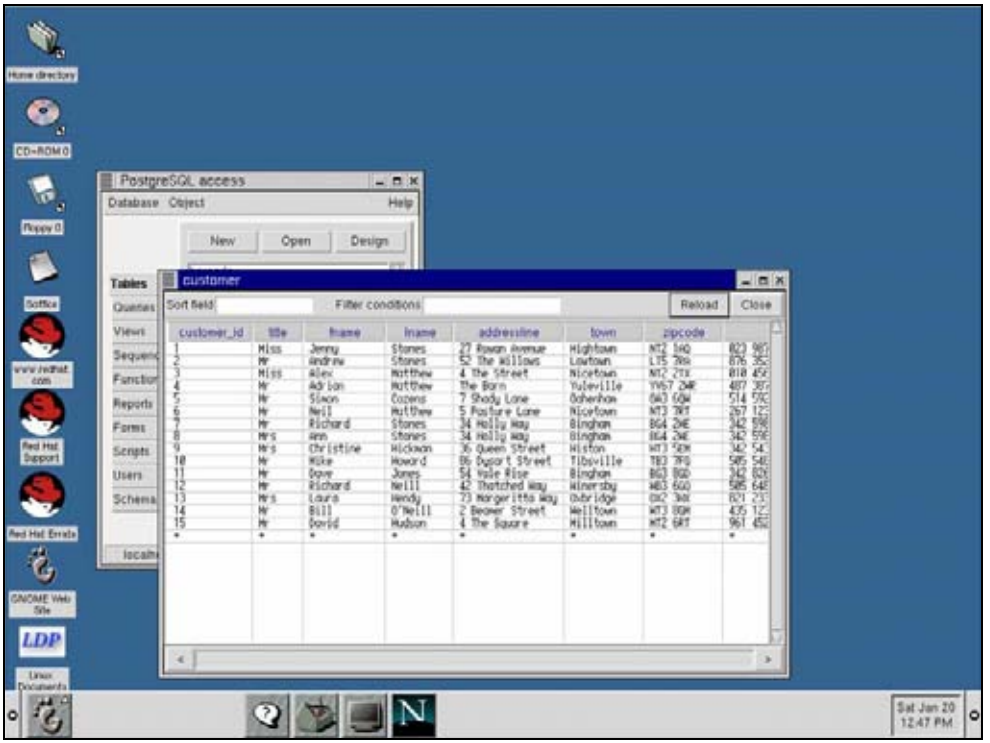


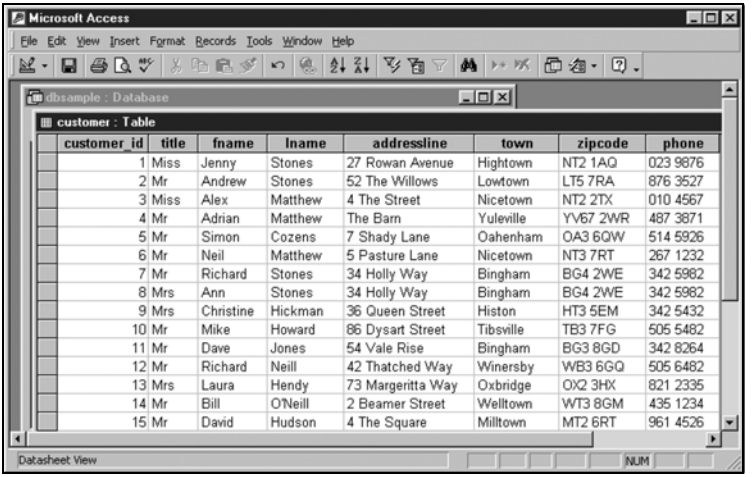
Рис. 2.5. Просмотр таблицы PostgreSQL с помощью pgAccess

Сетевой доступ

Конечно же, если бы доступ к данным был возможен только с той машины, на которой они хранятся, ситуация практически не отличалась бы от той, когда существует обычный файл, доступ к которому разрешен нескольким пользователям.

PostgreSQL – это база данных на выделенном сервере, и, как было упомянуто ранее, после соответствующей настройки она принимает запросы от клиентов по сети. Естественно, запрос может поступить и с той же машины, на которой работает сервер базы данных, но это маловероятно при многопользовательском доступе. Под Microsoft Windows можно применять драйвер ODBC. Таким образом можно по сети связать любое настольное приложение под Windows, поддерживающее ODBC, с сервером, хранящим наши данные. Технические подробности этого процесса будут описаны в главе 4.

На рис. 2.6 представлена база данных MS Access, содержащая связанные внешние таблицы (по ODBC) для доступа к нашей базе данных PostgreSQL, установленной на машине с операционной системой Linux:



customer_id	title	fname	lname	addressline	town	zipcode	phone
1	Miss	Jenny	Stones	27 Rowan Avenue	Hightown	NT2 1AQ	023 9876
2	Mr	Andrew	Stones	52 The Willows	Lowtown	LT5 7RA	876 3527
3	Miss	Alex	Matthew	4 The Street	Nicetown	NT2 2TX	010 4567
4	Mr	Adrian	Matthew	The Barn	Yuleville	YV67 2WR	487 3871
5	Mr	Simon	Cozens	7 Shady Lane	Oahenham	OA3 6QW	514 5926
6	Mr	Neil	Matthew	5 Pasture Lane	Nicetown	NT3 7RT	267 1232
7	Mr	Richard	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982
8	Mrs	Ann	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982
9	Mrs	Christine	Hickman	36 Queen Street	Histon	HT3 5EM	342 5432
10	Mr	Mike	Howard	86 Dysart Street	Tibsville	TB3 7FG	505 5482
11	Mr	Dave	Jones	54 Vale Rise	Bingham	BG3 8GD	342 8264
12	Mr	Richard	Neill	42 Thatched Way	Winersby	WB3 6GQ	505 6482
13	Mrs	Laura	Hendy	73 Margeritta Way	Oxbridge	OX2 3HX	821 2335
14	Mr	Bill	O'Neill	2 Beamer Street	Welltown	WT3 8GM	435 1234
15	Mr	David	Hudson	4 The Square	Milltown	MT2 6RT	961 4526

Рис. 2.6. База данных MS Access

Итак, доступ к одним и тем же данным может одновременно осуществляться по сети с нескольких машин. Данные при этом существуют в единственном экземпляре и находятся в надежном хранилище – на центральном сервере, который доступен по сети нескольким настольным компьютерам, работающим под управлением различных операционных систем. Как и все реляционные СУБД, PostgreSQL автоматически гарантирует невозможность конфликта обновлений данных в базе. Пользователям кажется, что все они имеют неограниченный доступ ко всем данным, на самом же деле «за кулисами» PostgreSQL следит за обновлениями и предотвращает конфликты.

Способность предоставлять многим пользователям доступ к данным и при этом всегда гарантировать непротиворечивость данных является одним из важнейших свойств СУБД. Когда кто-то вносит изменения в столбец, мы видим либо то, что было до начала редактирования, либо его конечный результат, но никогда не получим частично измененных данных.

Классическим примером является банковская база данных для перевода денег с одного расчетного счета на другой. Если во время перевода денег кто-то пытается вывести отчет о состоянии балансов всех счетов, чрезвычайно важно, чтобы итоговая сумма была правильной. Для отчета не имеет значения, на каком из двух счетов находились деньги в момент запуска отчета, важно только, чтобы в нем не отразилось то промежуточное состояние, когда с одного счета сумма уже снята, а на другой еще не поступила.

Реляционные СУБД, и в том числе PostgreSQL, скрывают все промежуточные состояния, так что другие пользователи не могут их увидеть. Возникает понятие **изоляция** – отчет о состоянии балансов изолирован от операции перевода денег, поэтому он выполняется или до, или после нее, но не в тот же самый момент. В главе 9 об изоляции будет рассказано подробнее.

Выборка данных

Теперь, когда мы узнали, что не составляет труда получить доступ к данным, хранящимся в базе, поговорим о том, как же это сделать. Обычно к большим объемам данных применяются две основные операции.

Первая – это выборка строк, соответствующих некоторому набору значений, а вторая – выборка подмножества столбцов. В терминах баз данных такие операции называются выборкой и проекцией соответственно, эти понятия чуть более формальны, но все так же просты и понятны.

Рассмотрим выбор подмножества строк. Предположим, что нам нужна информация обо всех клиентах, живущих в городе Бингаме (Bingham). Используем `psql`, стандартную утилиту командной строки PostgreSQL, для того чтобы показать, как применять язык SQL для обращения к серверу с целью получения нужных данных.

Команда SQL, необходимая в данном случае, выглядит очень просто:

```
SELECT * FROM customer WHERE town = 'Bingham';
```

Если команда вводится в `psql` или каком-то другом клиентском приложении с графическим интерфейсом, следует добавить точку с запятой, чтобы обозначить конец команды (т. к. существуют длинные команды, занимающие несколько строк). Как правило, в этой книге будут ставиться точки с запятой, поскольку для тех, кто будет выполнять примеры, используя графический интерфейс, ввод точки с запятой обязателен.

PostgreSQL в ответ на команду возвращает все строки таблицы клиентов, в которых в столбце, отведенном для города, указано «Bingham» (рис. 2.7):

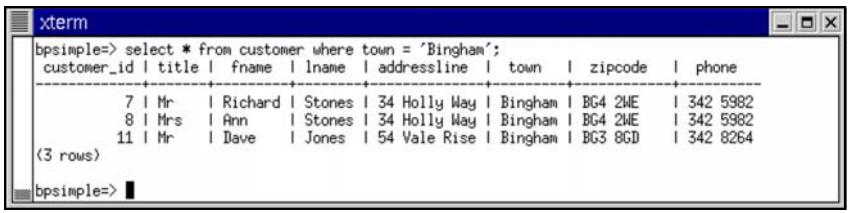


Рис. 2.7. Пример выборки подмножества строк

Как видите, все было очень просто. Пока не думайте о конкретном виде операторов SQL, вернемся к этому в главе 5. Обратите внимание на добавленный столбец `customer_id`, он будет использован позднее, когда в базе данных будут сохраняться заказы.

Итак, это была выборка, при которой из таблицы отбирались определенные строки. Теперь займемся проекцией, или, проще говоря, выбором, отдельных столбцов из таблицы.

Пусть нас интересуют только имена и фамилии клиентов из таблицы `customer`. Столбцы с этими данными были названы `fname` и `lname`. Команда для извлечения указанных столбцов также чрезвычайно проста:

```
SELECT fname, lname FROM customer;
```

Получив команду, PostgreSQL возвращает соответствующие столбцы (рис. 2.8):

Рис. 2.8. Пример выборки подмножества столбцов (проекции)

Разумно было бы предположить, что иногда бывает необходимо выполнить обе операции с данными одновременно, то есть выбрать конкретные столбцы и только из определенных строк. В SQL и это нетрудно. Например, пусть нам нужны имена и фамилии тех клиентов, которые живут в городе Бингаме (Bingham). Просто объединим два оператора SQL в одной команде:

```
SELECT fname, lname FROM customer WHERE town = 'Bingham';
```

PostgreSQL в ответ на запрос возвращает выбранные столбцы (рис. 2.9):

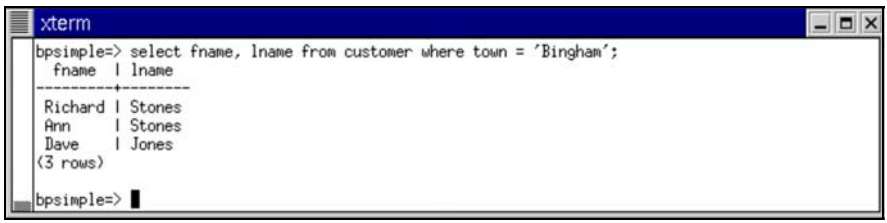


Рис. 2.9. Одновременная выборка строк и столбцов

Есть одна вещь, на которую стоит обратить внимание. Во многих традиционных языках программирования, таких как C или Java, нам пришлось бы написать некоторое количество кода для просмотра всех строк таблиц, остановки при нахождении строки с нужным городом и вывода запрошенных имен и фамилий. Даже если бы и удалось уместить все это в одну строку кода, эта строка была бы очень длинной и сложной. Дело в том, что C, Java и другие подобные языки по существу являются процедурными. В них приходится указывать, как компьютер должен себя вести. А в SQL, который является декларативным языком, надо просто сказать машине, что требуется, и внутренняя магия PostgreSQL сама справится с решением.

Тем, кто никогда раньше не имел дело с декларативным языком, ситуация может показаться немного странной, но если привыкнуть к этому, станет очевидно, что просто рассказать компьютеру о том, чего вы хотите, гораздо удобнее, чем объяснять ему, как это сделать. Вы даже будете удивлены, что раньше могли жить без таких языков.

Добавление информации в базу данных

Если бы реляционные базы данных могли использоваться только так, как было только что описано, вряд ли появился бы стимул заменить ими электронные таблицы. Однако далее будет рассказано о том, что реляционные СУБД, такие как PostgreSQL, обладают гораздо более широкими возможностями.

Несколько таблиц

Следующее свойство, которое мы рассмотрим, призвано решить проблему с добавлением информации о заказах клиентов (электронная таблица становилась не очень-то аккуратной, когда мы пытались вводить информацию о заказах в строке каждого клиента). Как же хранить данные о заказах, если заранее не известно, сколько заказов может сделать каждый клиент?

По названию раздела вы, наверное, уже догадались, что в случае реляционной базы данных решением является добавление второй таблицы, которая и будет содержать дополнительную информацию. Так же как при проектировании таблицы `customer`, начнем с того, что определим, какого рода данные необходимо хранить для каждого заказа.

Пока предположим, что надо хранить имя клиента, разместившего заказ, дату размещения, дату отгрузки и расходы на доставку. Как и в таблицу `customer`, введем уникальный номер для ссылок, чтобы каждый заказ отличался от всех остальных (вместо того, чтобы пытаться угадать, какой признак однозначно характеризует заказ). Что касается информации о клиенте, естественно, нет необходимости повторно

сохранять все данные. Все сведения о клиенте, находящиеся в таблице `customer`, могут быть получены, если известен `customer_id`.

Может показаться, что стоит хранить детали заказа, – в конце концов, для большинства клиентов этот аспект является немаловажным, они ведь хотят получить то, что заказали! Но мы пока не будем хранить такие данные, позже станет ясно почему. Самые толковые читатели, наверное, уже догадались, что дело все в той же проблеме повторяющихся групп – заранее не известно, сколько деталей есть в каждом заказе (так же, как не известно, сколько заказов сделает каждый клиент). Несомненно, так оно и есть, но давайте двигаться постепенно!

Таблица с информацией о заказах, содержащая некоторые данные, представлена (рис. 2.10) в графическом средстве `pgAccess`, которое, как было показано ранее, позволяет получить доступ к данным в базе данных PostgreSQL из графического интерфейса. В главе 4 о графических интерфейсах рассказано подробно:

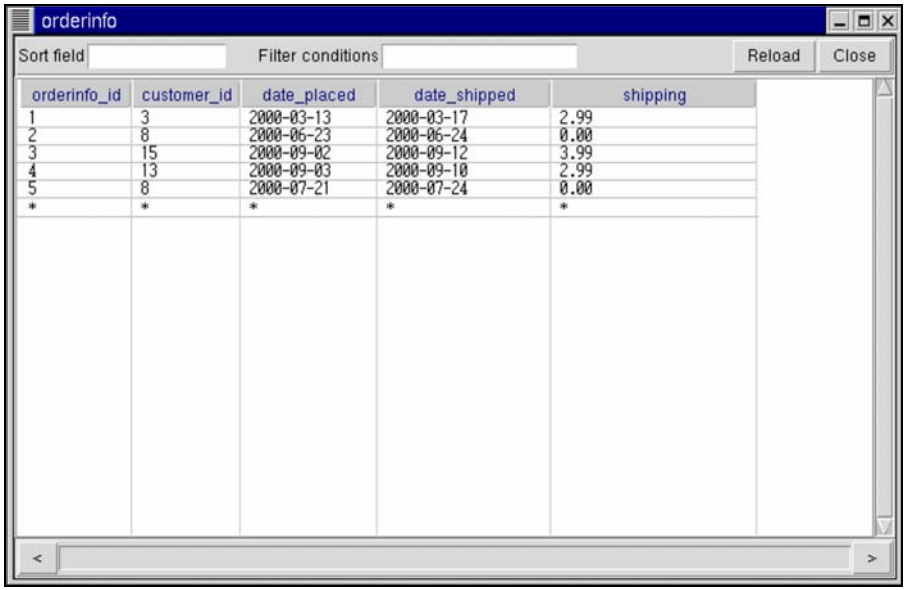


Рис. 2.10. Таблица заказов

В таблицу специально помещено не слишком много данных, т. к. экспериментировать удобнее с небольшими объемами.

Отношения между таблицами

Теперь в нашем распоряжении есть подробные данные о клиентах и, как минимум, сводные данные об их заказах, хранящиеся в базе данных. Во многих отношениях нынешняя ситуация ничем не отличается

от использования двух электронных таблиц, одной – для клиентов, а другой – для заказов. Пришло время посмотреть, как можно использовать сочетание двух таблиц. Для этого осуществим одновременную «выборку» данных из обеих таблиц. Такая операция называется объединением и является третьей (после выборки и проекции), самой популярной операцией извлечения данных в SQL.

Предположим, что требуется получить перечень всех заказов и разместивших их клиентов. При использовании процедурного языка, такого как C, пришлось бы написать программу, которая просматривала бы одну таблицу, вероятно, сначала `customer`, а затем для каждого интересующего нас клиента вывести все размещенные им заказы. Это несложно, но поглощает немало времени, к тому же кодирование достаточно утомительно. Вам наверняка будет приятно узнать, что в SQL сделать это гораздо проще благодаря операции объединения. От пользователя требуется лишь ответить на три вопроса:

- Какие столбцы его интересуют?
- Из каких таблиц нужно извлекать данные?
- Как две таблицы связаны между собой?

По-прежнему не стоит уделять особого внимания конкретному виду операторов SQL. Команда, которая потребуется в данном случае, приводилась в качестве примера в предыдущей главе:

```
SELECT * FROM customers, orderinfo WHERE customer.customer_id =
orderinfo.customer_id;
```

Как несложно догадаться, команда содержит запрос на вывод всех строк из наших двух таблиц, а также сообщает SQL, что столбец `customer_id` таблицы `customer` (обозначение *таблица.столбец* позволяет указать как имя таблицы, так и название столбца внутри нее) содержит ту же информацию, что и `customer_id` в таблице `orderinfo`.

Теперь, когда в базе данных уже несколько таблиц с данными, посмотрим, как PostgreSQL отреагирует на ввод команды (рис. 2.11):

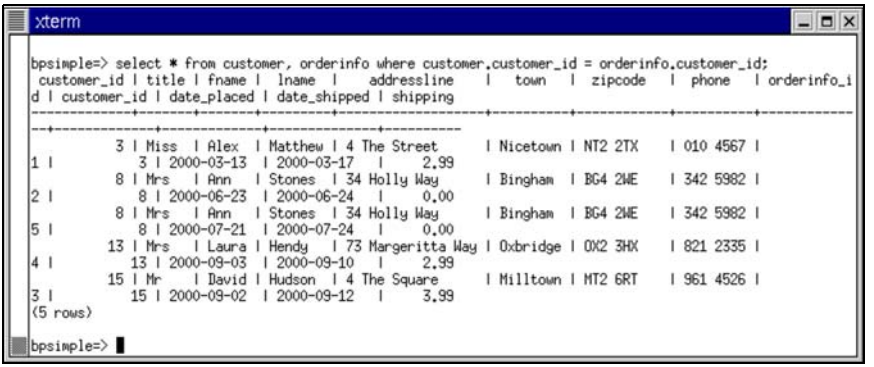


Рис. 2.11. Пример использования операции объединения

Выглядит не очень аккуратно, поскольку приходится переносить строки для того, чтобы не выходить за границы экрана, однако все же можно понять, как PostgreSQL ответила на запрос (притом, что не было точных указаний относительно того, как должна быть решена эта задача). Символ «*» можно заменить названиями столбцов, если требуется выбрать лишь указанные столбцы (например, если нас интересуют только фамилии и суммы заказов).

Теперь забежим немного вперед и рассмотрим гораздо более сложный запрос, касающийся этих двух таблиц, который можно выполнить при помощи SQL. Пусть требуется узнать, как часто различные клиенты размещали заказы. Здесь потребуется более глубокое знание SQL, команда же будет выглядеть так:

```
SELECT customer.title, customer.fname, customer.lname,
count(orderinfo.orderinfo_id) AS "Number of orders" FROM customer, orderinfo
WHERE customer.customer_id = orderinfo.customer_id group by customer.title,
customer.fname, customer.lname;
```

Выглядит сложнее, чем то, что встречалось ранее, но если не вдаваться в детали, видно, что, опять-таки, мы не говорим, как решить задачу, а просто очень подробно описываем ее, используя SQL. При этом все умещается в одном операторе. Вот как отвечает PostgreSQL (рис. 2.12):

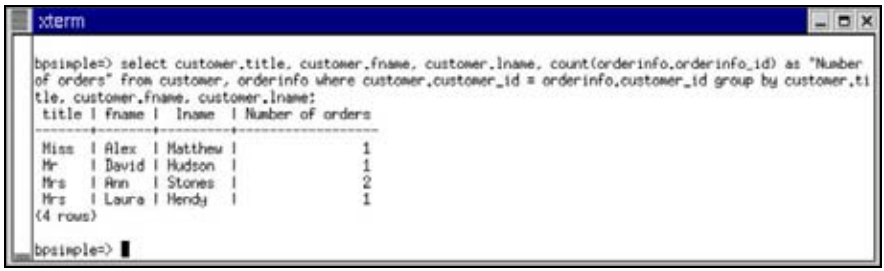


Рис. 2.12. Более сложный пример использования операции объединения

Некоторым специалистам баз данных может нравиться вводить операторы SQL в окне утилиты командной строки (а в некоторых случаях без нее и не обойтись), но надо признать, что у пользователей могут быть и другие предпочтения. Например, если кому-то нравится графический интерфейс Windows, то никаких трудностей это не создаст, доступ к базе данных можно будет получить при помощи драйвера ODBC, а запросы создавать графически. Далее (рис. 2.13) представлен тот же самый запрос, реализованный в StarOffice на машине с операционной системой Windows NT (StarOffice Base представляет собой альтернативу СУБД Microsoft Access в Windows).

Данные все так же хранятся на машине под управлением Linux, но пользователю едва ли необходимо знать технические подробности. Как правило, при изучении SQL в этой книге будет рассматриваться

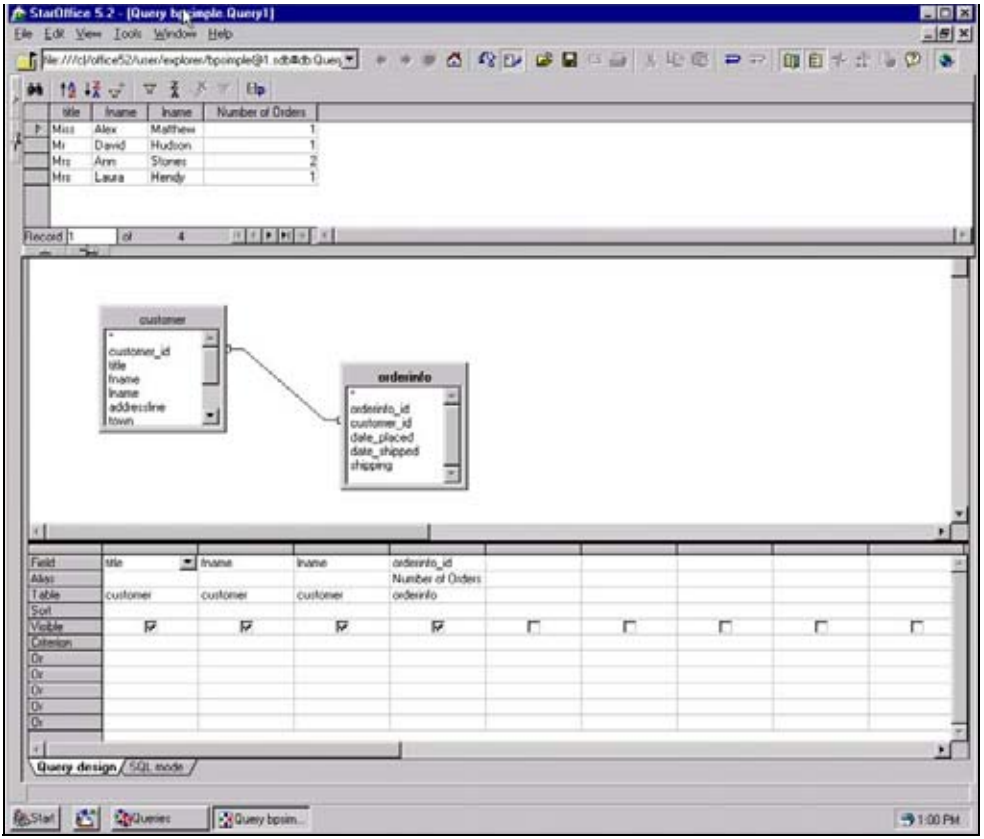


Рис. 2.13. Запрос, реализованный в StarOffice

командная строка, потому что так читатели смогут выучить основы, прежде чем придет время переходить к более сложным командам SQL, работа с которыми при помощи средств графического интерфейса не всегда удобна. Конечно, ваше право предпочесть средствам командной строки ввод команд SQL в графическом интерфейсе, – поступайте так, как вам удобнее.

Проектирование таблиц

Пока в нашей базе данных всего две таблицы, и до сих пор мы еще не говорили о том, как принимается решение о том, какие данные попадут в каждую из таблиц; мы делали то, что представлялось нам разумным. На самом деле проектирование схемы базы данных (корректно называть конструкцию, включающую в себя таблицы, столбцы и отношения, схемой) может быть очень сложным занятием. Основные этапы этого процесса, а также правила, которые нужно соблюдать, будут описаны в главе 12.

Проектирование базы данных, состоящей из нескольких десятков таблиц, представляет собой серьезную задачу. Люди, занимающиеся этим, получают деньги за то, что они хорошо делают эту трудную работу.

Несколько эвристических правил

К счастью, для достаточно простых баз данных (приблизительно до десяти таблиц) проектирование не так уж сложно, и вы, приобретя некоторые навыки, сможете создавать разумные конструкции.

В этом разделе рассмотрим простую базу данных, которую мы уже начали создавать, и попытаемся понять, какими соображениями следует руководствоваться, определяя, какие таблицы необходимо создать.

При проектировании базы данных обычно выполняется «нормализация», то есть применяется некоторый набор правил, гарантирующих, что данные будут распределены соответствующим образом. В главе 12 будет представлено более формальное описание данного вопроса. Для начала же нам потребуется несколько основных правил. Надеемся (и настаиваем на том, что этого не следует делать), что вам не придет в голову, прочитав эти правила, сразу же взяться за создание базы, состоящей из двадцати таблиц, прочитайте книгу до конца или, по крайней мере, до главы 12.

Эти правила приведены только для того, чтобы облегчить читателям понимание базы данных, которая используется в этой книге для изучения SQL и PostgreSQL.

Правило первое – разбивайте данные на столбцы

Первое правило состоит в том, что в каждый столбец должен быть помещен один фрагмент информации (атрибут). Это кажется вполне логичным. Уже в электронной таблице информация по каждому клиенту была естественным образом разбита на столбцы так, что, например, фамилия была отделена от почтового индекса.

В электронных таблицах соблюдение этого правила делает более удобной работу с данными, например сортировку по индексу. Что же касается баз данных, важнейшим условием является правильное разбиение данных на атрибуты. Почему это так важно? С практической точки зрения весьма сложно будет указать, что нас интересуют данные, находящиеся с 29-й по 35-ю позицию в столбце адреса, т. к. именно там должен находиться почтовый индекс. Если данных достаточно много, наступит момент, когда индекс появится на другой позиции, и мы получим неверный фрагмент информации. Еще одна причина необходимости корректного разбиения на столбцы состоит в том, что все столбцы базы данных должны содержать данные одного типа.

Правило второе – однозначно идентифицируйте каждую строку

Если вы помните, при попытке однозначно идентифицировать строки в электронной таблице оказалось, что трудно решить, что же является уникальным признаком. Не было первичного ключа. Первичный ключ не обязательно должен представлять собой отдельный уникальный столбец, это может быть и комбинация двух и даже трех столбцов, которая однозначно определяет строку.

В любом случае необходимо задать что-то так, чтобы, используя это что-то, можно было с полной уверенностью сказать: «Глядя на X в этой строке, можно не сомневаться, что его значение будет отличаться от значений всех остальных строк данной таблицы». Если не удается выбрать столбец или комбинацию столбцов, которая однозначно определяла бы строку, необходимо ввести дополнительный столбец, смысл которого будет только в обеспечении уникальности строки. В таблицу `customer` был введен дополнительный столбец `customer_id`, и теперь строки идентифицируются однозначно.

Правило третье – удаляйте повторяющуюся информацию

Когда мы пытались сохранить данные о заказах в таблице `customer`, она становилась некрасивой из-за проблемы повторяющихся групп. Для каждого клиента приходилось повторять информацию о заказе столько раз, сколько требовалось. То есть предугадать, сколько столбцов нужно отвести для заказов, было невозможно. В базе данных количество столбцов фиксируется при проектировании. Так что необходимо заранее решить, сколько потребуется столбцов, какого они будут типа, и дать каждому столбцу название, только тогда можно будет сохранять какие-то данные. Ни в коем случае не храните повторяющиеся группы данных в одной строке.

Обойти это ограничение можно так, как мы это и сделали, – разделив данные о клиентах и заказах на две таблицы и используя столбец `customer_id` для объединения таблиц в тех случаях, когда потребуется информация из обеих таблиц.

Если говорить более формально, была создана зависимость «многие-к-одному», другими словами, несколько заказов могло быть получено от одного клиента.

Правило четвертое – давайте непротиворечивые названия

Вероятно, это правило сложнее всего выполнить. Как назвать таблицу или столбец? Если название не придумывается, это часто бывает сигналом того, что в конструкции таблицы или столбца что-то неладно.

Многие разработчики баз данных имеют свой собственный набор правил, которыми они руководствуются, именуя таблицы и столбцы для того, чтобы в названиях не было противоречивости. Например, нельзя, чтобы имена каких-то таблиц были существительными в единственном числе, а других – во множественном (не «офис» и «отделы», а «офис» и «отдел»). Если выбрано имя для столбца с идентификаторами, может быть, `tablename_id`, то следует придерживаться этого правила и в названиях других подобных столбцов. Если вы используете сокращения, делайте это постоянно. Если столбец одной таблицы является ключом для другой (см. главу 12), постарайтесь дать им имена, образованные от одной основы.

Цель всех этих действий чрезвычайно проста – таблицы и столбцы должны иметь короткие значащие имена, а процесс присваивания имен в пределах базы данных должен быть последовательным. Достижение такой, казалось бы, простой цели, часто бывает весьма непростым занятием, но и вознаграждение мы получаем не маленькое – ведение базы данных значительно облегчается.

Схема базы данных «Клиенты и заказы»

Можно нарисовать план (схему) базы данных, отразив отношения сущностей. Для нашей базы данных из двух таблиц схематическое изображение представлено на рис. 2.14:

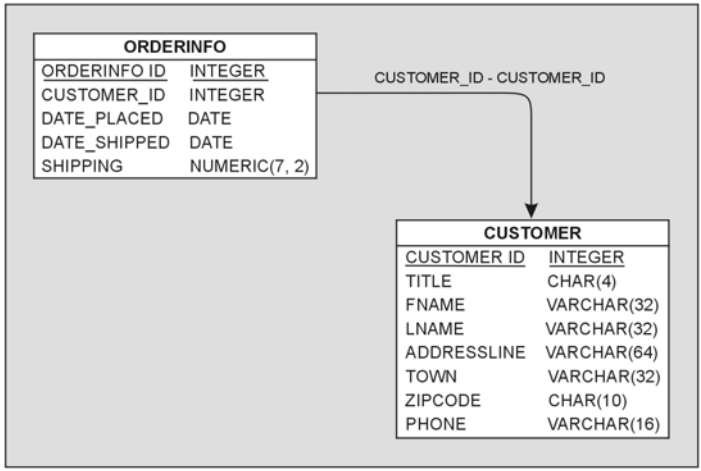


Рис. 2.14. База данных «Клиенты и заказы»

Представлены две таблицы, столбцы, типы данных и размеры для каждого столбца, а также показано, что `customer_id` – это столбец, объединяющий две таблицы. Обратите внимание на направление стрелки – из таблицы `orderinfo` в таблицу `customer`. Это означает, что

для каждой записи в `orderinfo` есть не более одной записи в `customer`, но каждому клиенту может соответствовать несколько заказов.

Очень важно помнить о том, кто есть кто в отношении «один-ко-многим», иначе может возникнуть масса проблем. Заметьте также, что столбец, который будет использоваться для объединения двух таблиц, получил одно и то же имя в обеих таблицах – `customer_id`. Это не жизненно важно, при желании можно было назвать эти два столбца `foo` и `bar`, но постоянство в именовании в дальнейшем очень нам поможет.

В сложных базах данных отсутствие единообразия в названиях (например, `customer_id`, `customer_ident`, `cust_id` или `cust_no`) может быть чрезвычайно неудобным. Поэтому всегда разумнее потратить время на присваивание хороших и осмысленных имен, это надолго облегчит вам жизнь.

Добавляем таблицы в базу данных

Очевидно, что информация, в данный момент находящаяся в базе данных, является недостаточной, поскольку не известно, из каких пунктов состоит каждый заказ. Если помните, составляющие заказов были опущены сознательно, при этом мы обещали вернуться к этой проблеме позже. Пришло время распределить товары по заказам.

Проблема состоит в том, что заранее не известно, сколько пунктов содержит каждый заказ (аналогично тому, как заранее не известно, сколько заказов может разместить каждый из клиентов). Заказ может включать в себя один, два или даже сто товаров. Необходимо отделить информацию о том, что клиент сделал заказ, от деталей самого заказа. Следует попытаться создать нечто, подобное изображенному на рис. 2.15:

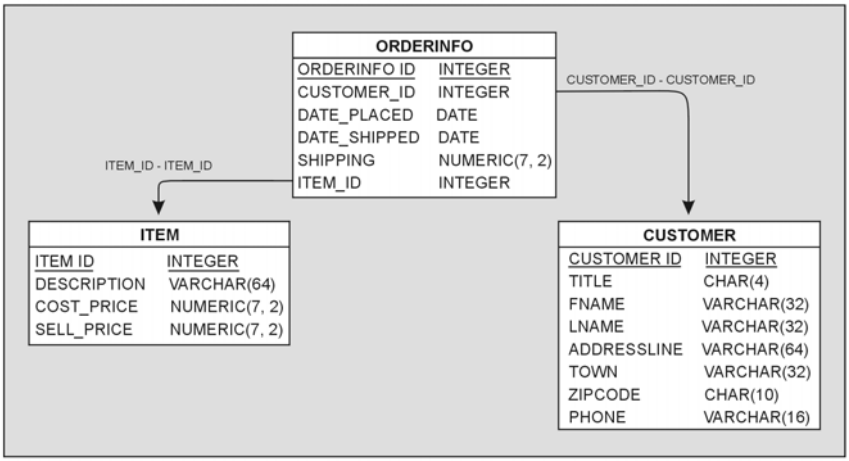


Рис. 2.15. Расширяем базу данных «Клиенты и заказы»

Как и при создании таблиц `customer` и `orderinfo`, разделим информацию на две таблицы, а затем объединим их.

Правда при этом возникает одна небольшая проблема.

Если тщательно проанализировать отношения между заказом и отдельными товарами, станет ясно, что не только каждая запись из `orderinfo` может быть связана со многими товарами, но и один и тот же товар может присутствовать во многих заказах (если несколько клиентов заказали один и тот же товар). Попытаемся изобразить эти две таблицы на диаграмме сущностей (рис. 2.16):

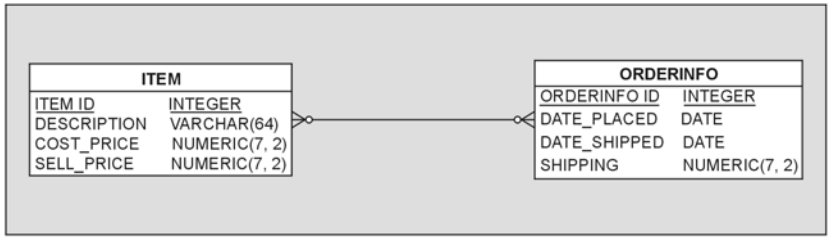


Рис. 2.16. Отношение между таблицами заказов и товаров

Представлено отношение «многие-ко-многим» – каждый заказ может ссылаться на множество товаров, и каждый товар может появляться во множестве заказов. Это подрывает некоторые из основ реляционных баз данных. Пока не будем вдаваться в детали, заметим только, что в реляционной базе данных никакие две таблицы не должны находится в отношениях «многие-ко-многим».

Можно попытаться обойти эту проблему, создав отдельную запись в каждой строке таблицы `item` для каждого заказа, но тогда придется многократно повторить описание и информацию о цене для каждого товара, что нарушило бы правило третье (удалять повторяющуюся информацию!).

Данный тип отношений будет рассмотрен в главе 12, а что касается нашей проблемы, то для нее существует стандартное решение. Необходимо создать третью таблицу между двумя существующими, она будет реализовывать отношение «многие-ко-многим». Надо сказать, что объяснять сложнее, чем делать, поэтому просто создадим эту третью таблицу, `orderline`, которая свяжет заказы с товарами.

Создана таблица, строки которой соответствуют каждой строчке заказа (рис. 2.17). Для каждой отдельной строки можно определить, из какого она заказа, по столбцу `orderinfo_id`, а товар, на который она ссылается, – по столбцу `item_id`.

Один товар может присутствовать во многих строках таблицы `orderline`, а один заказ также может встречаться в нескольких ее строках. Однако каждая строка промежуточной таблицы ссылается только на один товар и может присутствовать только в одном заказе.

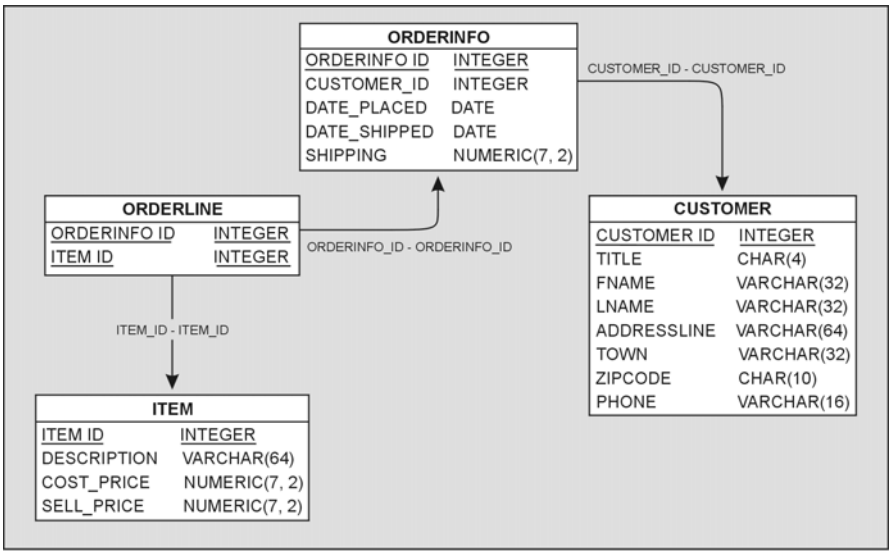


Рис. 2.17. Добавление в базу данных таблицы orderline

Обратите внимание, что в новую таблицу не введен уникальный идентификатор для каждой строки. Дело в том, что комбинация `orderinfo_id` и `item_id` всегда уникальна. Но и здесь есть одна почти незаметная проблема. Что произойдет, если клиент закажет два экземпляра товара в одном заказе?

Просто ввести еще одну строку в `orderline` нельзя, поскольку комбинация `orderinfo_id` и `item_id` должна быть уникальной. Неужели придется добавить еще одну специальную таблицу для обслуживания заказов, содержащих несколько экземпляров одного товара? К счастью, нет. Есть гораздо более простой способ. Надо просто ввести поле «количество» (`quantity`) в таблицу `orderline`, и все будет хорошо.

Завершение первичного проекта

Осталось сохранить еще две порции данных, и в первом приближении основная структура нашей базы данных будет готова. Нас интересуют штрих-код для каждого продукта, а также количество товаров каждого вида, имеющихся на складе.

Может случиться так, что каждый продукт будет иметь несколько штрих-кодов, потому что если производитель вносит значительные изменения в упаковку продукта, он также часто меняет и штрих-код. Например, все наверняка видели упаковки с надписью «+20% бесплатно», стоимость такой (например) бутылки лимонада не меняется, но зато благодаря этой рекламной акции лимонада вы получаете больше. Обычно в таких случаях производители изменяют штрих-код, но

сам продукт, по существу, не меняется. Возникает отношение: множество штрих-кодов для одного продукта. Добавляем таблицу для хранения штрих-кодов (рис. 2.18):

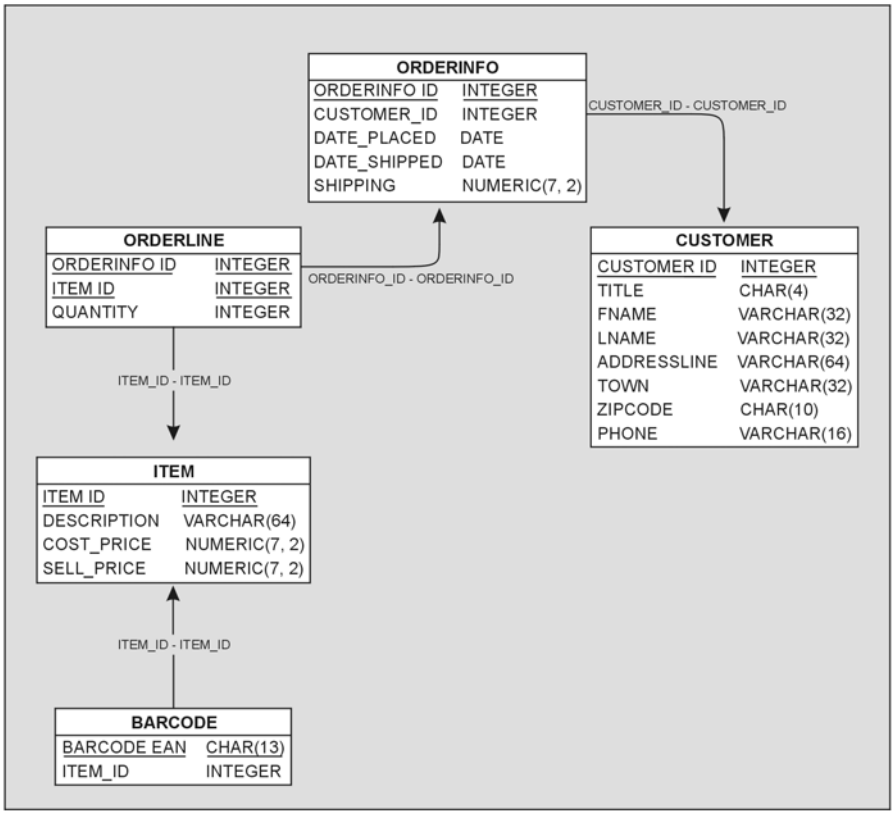


Рис. 2.18. Добавление в базу данных таблицы BARCODE

Заметьте, что стрелка идет в направлении от таблицы `BARCODE` к таблице `ITEM`, потому что именно штрих-кодов может быть несколько для одного товара. Обратите внимание на то, что `barcode_ean` является первичным ключом, т. к. для каждого штрих-кода должна существовать уникальная строка и (хотя один продукт и может иметь несколько штрих-кодов) ни один штрих-код не может принадлежать более чем одному продукту.

Наконец, последнее добавление, которое следует внести в проект базы данных, – объем запасов каждого продукта.

Есть два пути реализации такого добавления. Если большинство товаров находится на складе, то информация о таких запасах весьма важна, и тогда можно хранить количество продуктов, имеющихся на складе, непосредственно в таблице `item`.

Но может быть и так, что у нас множество товаров, при этом чаще всего лишь какие-то из них присутствуют на складе, а объем информации, которую приходится хранить для товаров, находящихся на складе, весьма велик. Например, для склада необходимо хранить информацию о размещении, номерах партий и сроках годности. Если в картотеке имеется 500 000 товаров, а на складе есть только 1000, то хранение данных по всем товарам будет просто расточительством. У этой проблемы есть стандартное решение – **дополнительная таблица**.

Следует создать новую таблицу, в которой хранилась бы «дополнительная информация» (например, сведения о хранении на складе), а затем создать только нужные строки – для продуктов, имеющихся в наличии на складе. Эти данные будут ссылаться на основную таблицу. На самом деле все гораздо проще, чем может показаться из этого объяснения. Окончательный проект первого варианта нашей базы данных, с которым мы будем работать далее, представлен на рис. 2.19:

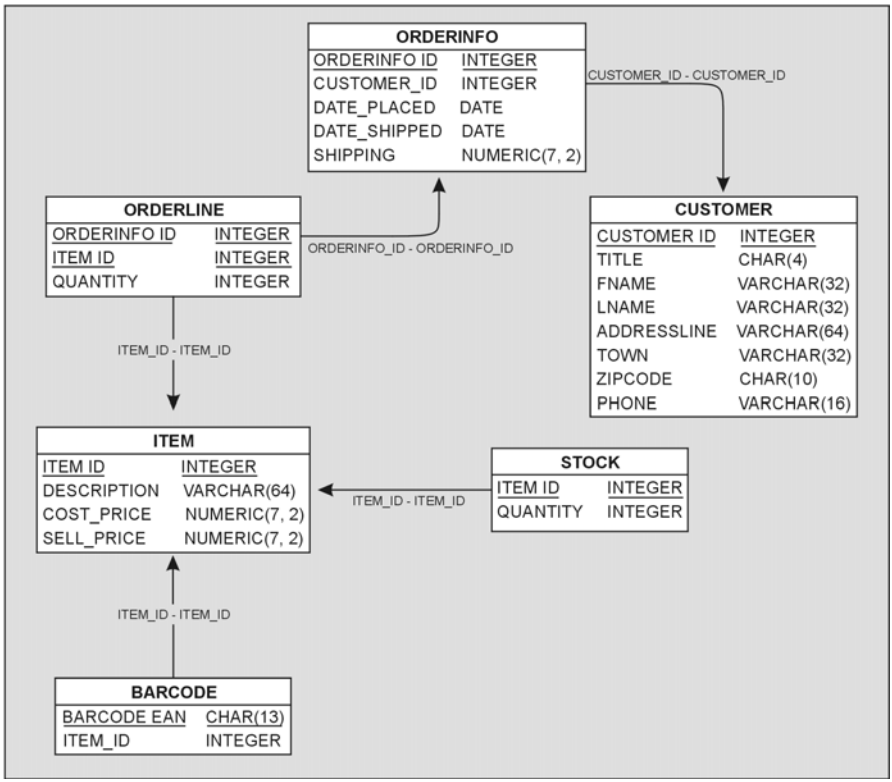


Рис. 2.19. Добавление в базу данных таблицы stock

Обратите внимание , что в таблице `stock` в качестве уникального ключа выступает `item_id`, а хранящаяся в ней информация связана непосредственно с товарами, при этом для выполнения соединения с соответ-

ствующим товаром в таблице item используется item_id. Стрелка указывает на таблицу item, потому что это главная таблица, хотя в данном случае и нет отношения «многие-к-одному».

В таком виде схема кажется явно запутанной, ведь хранимой нами дополнительной информации очень мало. Но оставим все как есть для того, чтобы показать, как это делается, а далее в книге расскажем, как получить доступ к данным в случае, если много информации содержится в дополнительных таблицах (таких, как данная). Для тех, кто любит заглядывать вперед, скажем, что будет использоваться «внешнее объединение».

Отметьте также, что названия некоторых столбцов на рисунке подчеркнуты, это означает, что данный столбец или комбинация столбцов (например, в таблице orderinfo) гарантированно уникальны. Они образуют первичный ключ таблицы.

Основные типы данных

До этого момента разговор о типах данных велся в общих словах. Из схемы видно, что использованные типы (далеко не все из имеющихся в PostgreSQL) – это скорее базовые типы, известные средству проектирования, которые могут быть преобразованы в реальные типы данных PostgreSQL при создании реальных таблиц.

Перед нами осталось еще одно препятствие, преодолением которого мы пока не занимались, – генерирование уникальных ключей таких полей, как customer_id и item_id. Вы, конечно же, помните, что каждая строка таблицы должна быть однозначно идентифицируемой и что в тех случаях, когда нет очевидного набора столбцов, который мог быть использоваться для достижения этой цели, добавляется дополнительный столбец, столбец с уникальным идентификатором. На ум приходит поле целых чисел или символьное поле, при этом, добавляя каждую новую строку, необходимо генерировать новое уникальное значение для столбца.

Поскольку необходимость введения дополнительного уникального столбца встречается при проектировании баз данных повсеместно, существует и встроенное решение, новый тип данных SERIAL. Этот специальный тип представляет собой целое число, автоматически увеличивающееся при добавлении строки в таблицу и устанавливающее новое уникальное значение при добавлении каждой строки. Если новая строка вводится в таблицу, имеющую столбец типа SERIAL, не нужно указывать никакого значения для этого столбца, СУБД сама автоматически присвоит следующее значение.

В большинстве баз данных при присваивании последовательных величин удаленные строки не учитываются. Присваиваемое число просто увеличивается для каждой новой строки. Средство проектирования, с

помощью которого были выполнены представленные выше диаграммы, показывает поля типа SERIAL как INTEGER, потому что этот тип является базовым.

Подытожим типы данных, использованные в данной схеме, в табл. 2.1.

Таблица 2.1. Типы данных

Тип данных	Описание
INTEGER	Целое число.
SERIAL	Целое число, автоматически устанавливаемое в уникальную величину для каждой добавляемой строки. На рисунках это не отражено, но именно этот тип будет использоваться для столбцов «_id».
CHAR	Символьный массив фиксированного размера, размер указывается в скобках после названия типа. В столбцах данного типа PostgreSQL всегда будет сохранять ровно указанное количество символов. Если CHAR(256) служит для хранения одного символа, в базе данных будет занято как минимум 256 байт, которые и будут возвращены при извлечении данных.
VARCHAR	Это также символьный массив, но (как понятно из названия) переменной длины, и обычно место, занимаемое в базе данных, практически совпадает с фактическим размером сохраняемых данных. При обращении к полю VARCHAR возвращается то количество символов, которое и было сохранено. В скобках после названия типа указывается максимально возможная длина. Естественно возникает вопрос о том, зачем нужны оба эти типа, почему нельзя ограничиться одним VARCHAR? Дело в производительности. Записи фиксированной длины база данных может обрабатывать гораздо быстрее, чем записи переменной длины. Поэтому если, например, известно, что размер данных в столбце не больше четырех символов, то лучше всегда сохранять четыре символа, а не заставлять базу данных каждый раз определять размер, т. к. места выиграно немного, а вот производительность для переменной длины обычно уменьшается.
DATE	Позволяет хранить данные о годе, месяце и дне. Конечно же, есть и другие родственные типы, позволяющие хранить данные о времени (вместе с информацией о дате или без). Вернемся к этому позже.
NUMERIC	Возможность хранить числа с указанным количеством разрядов (первое число в скобках) и фиксированным количеством разрядов после запятой (второе число в скобках). Так, NUMERIC(7,2) сохранит ровно семь разрядов, два из них – после запятой.

Далее в книге будут рассмотрены и другие типы данных PostgreSQL, при этом окажется, что какие-то из них больше подходят для хранения денежных значений, чем NUMERIC, поскольку несмотря на то, что использовать NUMERIC весьма удобно, это не самый эффективный способ хранения чисел с плавающей запятой в PostgreSQL.

Значение NULL

Новичков в теории баз данных может смутить такое понятие, как NULL. В терминах баз данных NULL обычно означает, что величина не известна (хотя существуют еще одна-две разновидности определения с едва уловимыми отличиями).

Посмотрим на таблицу orderinfo, в ней содержится столбец дат заказа, а также столбец дат отгрузки, оба имеют тип DATE. Как следует поступить, если заказ уже получен, но еще не отправлен? Что сохранить в столбце дат отгрузки? Можно сохранить специальную дату, **сигнальную величину** (sentinel value), которая указывала бы, что заказ еще не отправлен. В системах UNIX можно использовать в качестве такой даты 1 января 1970 года, т. к. именно с этого дня системы UNIX ведут свой отсчет. В любом случае эта дата должна предшествовать дате создания базы данных, тогда будет очевидно, что дата специальная, в нашем же случае она будет иметь смысл «еще не отправлено».

Понятно, что такое решение далеко от идеала. Наличие специальных величин, разбросанных по таблицам, свидетельствует о неудачности проекта и часто приводит к ошибкам. Например, если к работе над проектом подключается новый программист, который не знает о присутствии специальных дат, он может заняться подсчетом того, сколько времени в среднем проходит между размещением заказа и его отгрузкой, и получить весьма неожиданные результаты (если в нескольких случаях даты отправления были установлены как предшествующие дате размещения заказа).

К счастью, во всех реляционных базах данных существует специальная величина под названием NULL, которая обычно означает «в настоящий момент неизвестно». Обратите внимание, что это не ноль, не пустая строка и не нечто, что могло бы быть представлено в поле некоторого типа. «Неизвестное» очень сильно отличается от 0 и пустой строки.

Чрезвычайно важно присматривать за такими величинами, потому что они могут появляться в произвольных местах, причиняя неудобства. В таблице orderinfo можно установить дату отправки в NULL (до того, как заказ отгружен), значение «в настоящий момент неизвестно» полностью соответствует нашей ситуации.

Существует и несколько другое значение NULL (оно не так распространено, как первое) – «не относится» (not relevant) к данной строке. Предположим, например, что проводится опрос группы людей, при этом один из вопросов касается цвета очков. Но для тех, кто очки не носит, этот вопрос, естественно, не имеет никакого смысла. В таком случае можно использовать NULL в соответствующем столбце, для того чтобы показать, что информация не имеет отношения к данной конкретной строке.

Проверка на NULL

Одним из свойств NULL является следующее: при сравнении двух значений NULL результатом всегда будет «не равны». Это может удивить, но подумайте, ведь NULL — это неизвестное, поэтому абсолютно логично, что при сравнении двух неизвестных оказывается, что они не равны.

В SQL предоставлена специальная возможность проверки величин на значение NULL, если необходимо найти их и проверить, то задается вопрос 'IS NULL'.

Поведение величин NULL может иметь свои особенности, поэтому при проектировании таблицы можно указать, что в некоторых столбцах не могут храниться NULL. Обычно удобно бывает сказать, что для столбцов будет выполняться условие 'NOT NULL', — для уверенности в том, что NULL там не появится (например, в столбце, представляющем собой первичный ключ). Некоторые проектировщики баз данных пропагандируют практически полную отмену использования величин NULL, но все же в них есть и определенная польза. Поэтому, как правило, поддерживается применение величин NULL в отдельных столбцах при наличии реальной потребности в значениях, которые не определены.

Образец базы данных

В данной главе была спроектирована (специальным образом) база данных для отслеживания клиентов, заказов и товаров, которая могла бы оказаться полезной в небольшом магазине. По ходу книги эта база данных будет использоваться для иллюстрации возможностей SQL и различных свойств PostgreSQL. Будет также рассказано об ограниченности существующей схемы и о том, как она может быть усовершенствована.

Эта упрощенная база данных содержит множество элементов, подобных тем, которые образуют настоящую базу данных для розничной торговли, но много и упрощений. Например, для товара может существовать полное описание, описание, которое касса печатает при его продаже, а также описание, представленное на этикетке.

Чрезвычайно упрощена и информация об адресе клиента. Длинный адрес, включающий в себя название деревни или штата, не может быть обработан, так же как и заграничный. Обычно бывает разумно начать с достаточно простой надежной базы данных, а затем уже наращивать ее, вместо того чтобы пытаться сразу же принять в расчет все возможные требования уже в первоначальном проекте. Для начала наша база данных вполне достаточна.

В следующей главе будет рассмотрена инсталляция PostgreSQL, создание таблиц для нашей базы и помещение в них некоторых данных.

Резюме

В этой главе рассказано о том, чем таблица базы данных похожа на обычную электронную таблицу, и об их важных отличиях:

- Все элементы столбца должны иметь один тип
- Количество столбцов должно быть одинаковым для всех строк
- Должна быть возможность однозначной идентификации каждой строки
- Никакого предписанного порядка строк не существует

Показано, как расширить базу данных до нескольких таблиц для удобства организации отношений «многие-к-одному». Приведены некоторые эвристические правила, помогающие понять, какой должна быть структура базы данных (в следующих главах поговорим об этом более формально).

Представлен способ реализовать отношения «многие-ко-многим», существующие в реальном мире, а именно разбить их на пару отношений «многие-к-одному», введя дополнительную таблицу.

Наконец, была проведена работа по расширению исходного проекта, благодаря чему у нас есть демонстрационный проект базы данных (схема), который будет усовершенствоваться далее.

В следующей главе будет рассказано о том, как поставить и запустить PostgreSQL на различных платформах.

3

Начинаем работу с PostgreSQL

В этой главе мы рассмотрим шаги, необходимые для установки PostgreSQL в различных операционных системах. Если вы работаете в системе Linux одной из последних версий, то, возможно, PostgreSQL уже установлен или имеется на установочных дисках. Для тех же, кто работает в UNIX, мы расс мотрим процесс компиляции исходного кода для платформы UNIX.

Мы также увидим, как установить PostgreSQL на платформе Windows, где перед инсталляцией PostgreSQL придется установить дополнительное программное обеспечение, моделирующее окружение UNIX.

По ходу дела коснемся следующих тем:

- Устанавливать или обновлять?
- Установка PostgreSQL из дистрибутива Linux
- Состав дистрибутива PostgreSQL
- Установка PostgreSQL из исходного кода и запуск PostgreSQL
- Создание баз данных
- Создание и заполнение таблиц
- Остановка PostgreSQL
- Установка PostgreSQL в Windows
- Cygwin – UNIX-среда в Windows
- Службы IPC в Windows
- Компиляция и конфигурирование PostgreSQL в Windows
- Автоматический запуск PostgreSQL в Windows

В настоящее время продолжается работа над версией PostgreSQL для Solaris. Пройдет еще некоторое время, прежде чем завершится ее окончательное тестирование. Поэтому установка PostgreSQL на Solaris не рассмотрена в этой книге. Энтузиасты PostgreSQL, желающие работать в Solaris, могут узнать последние новости по адресу *http://www.greatbridge.org* и даже принять участие в работе.

Устанавливать или обновлять?

Этот вопрос определенно возникнет у любителей Linux, решивших использовать PostgreSQL. В действительности на него нет однозначного ответа. Последняя версия на момент написания книги – PostgreSQL-7.1.2. Текущую версию можно получить с сайта *http://www.postgresql.org* и с перечисленных на нем ftp-серверов.

Если в вашем распоряжении имеется свежий дистрибутив Linux, то весьма вероятно, что в него входит и последняя версия PostgreSQL. Если же Linux установлен из более раннего дистрибутива, включающего версию 7.1.x, то можно обновить PostgreSQL до последней версии, загрузив соответствующий пакет с любого из сайтов, посвященных PostgreSQL. Однако в этой книге мы предполагаем, что читатель является новичком в PostgreSQL, и поэтому рассматриваем новую установку, а не трудоемкую и утомительную процедуру обновления.

Горячие поклонники Linux, предпочитающие обновления и имеющие в этом опыт, могут получить информацию по обновлению на сайте PostgreSQL.

В следующих разделах мы опишем установку PostgreSQL на UNIX-подобных платформах, на Linux и на Microsoft Windows. Процедура установки практически одинакова во всех разновидностях Linux.

Есть два основных способа установки PostgreSQL:

- установка PostgreSQL из дистрибутива Linux;
- установка PostgreSQL из исходного кода.

Мы начнем с установки PostgreSQL из дистрибутива Linux.

Установка PostgreSQL из дистрибутива Linux

Возможно, наиболее простой способ установки PostgreSQL на Linux состоит в использовании предварительно откомпилированных двоичных пакетов, доступных в формате менеджера пакетов RedHat (RedHat Package Manager, RPM) во многих дистрибутивах. Ко времени написания этой книги имелись пакеты RPM для следующих версий:

- RedHat 6.x и 7.x
- SuSe 6.4 и 7.x

- TurboLinux 6.x
- Caldera OpenLinux eServer 2.3
- Mandrake 7.x
- LinuxPPC 2000

Для установки полнофункциональной СУБД нам понадобятся как минимум базовый и серверный пакеты. Полный список пакетов приведен в табл. 3.1:

Таблица 3.1. Пакеты, необходимые для установки полнофункциональной СУБД

Пакет	Описание
postgresql	Базовый пакет
postgresql-server	Программы создания и запуска сервера
postgresql-devel	Заголовочные файлы и библиотеки для разработки
postgresql-jdbc	Драйверы JDBC для доступа к PostgreSQL из Java
postgresql-odbc	Драйверы ODBC для доступа к PostgreSQL
postgresql-perl	Интерфейс Perl для PostgreSQL
postgresql-python	Интерфейс Python для PostgreSQL
postgresql-tcl	Интерфейс Tcl для PostgreSQL
postgresql-test	Тестовые программы PostgreSQL
postgresql-tk	Оболочка и графический интерфейс Tk

К имени файла добавляется номер версии соответствующего пакета. Рекомендуется устанавливать пакеты с одинаковыми номерами версий и ревизий. Номер ревизии – это последняя цифра в номере версии, то есть x в 7.1.x.

Для установки пакетов используется программа RPM Package Manager. Перед тем как начинать установку, убедитесь, что вы зарегистрированы как суперпользователь (`superuser/root`). Для установки RPM-пакетов подойдет любой графический менеджер пакетов, например GnoRPM или Kpackage. Можно также скопировать все файлы RPM в один каталог и как суперпользователь (`superuser/root`) выполнить команду:

```
$ rpm -i *.rpm
```

Эта команда распаковывает пакет и копирует находящиеся в нем файлы в предназначенные для них каталоги.

Можно выполнить установку PostgreSQL и из пакета, входящего в состав дистрибутива Linux, такого как RedHat или SuSe. Например, в SuSe 7.x для установки PostgreSQL можно запустить утилиту YaST (рис. 3.1) и выбрать пакеты из списка, приведенного в табл. 3.1.

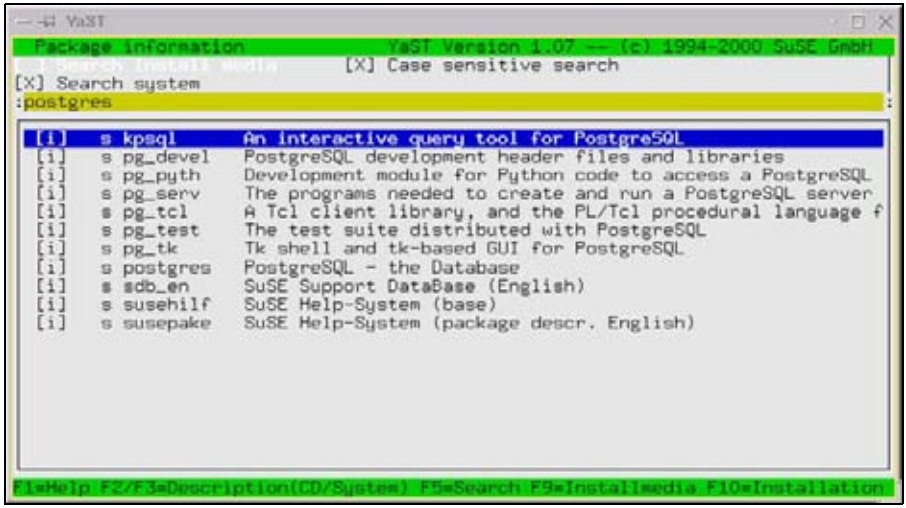


Рис. 3.1. Утилита установки YaST

В качестве альтернативы можно загрузить или заказать на CD коммерческую версию (само программное обеспечение остается бесплатным) PostgreSQL, включающую инсталлятор и дополнительную документацию. Одна из таких коммерческих версий доступна на сайте *http://www.greatbridge.com* компании Great Bridge.

Разработка PostgreSQL не прекращается, и, следовательно, время от времени будут выходить новые версии. Установка «вручную» из пакетов RPM имеет то преимущество, что позволяет обновлять установленную версию до текущей, просто повторяя процедуру. Для этого достаточно сообщить программе установки, что она должна выполнить обновление, а не первоначальную установку, заменив параметр `-i` на `-u`:

```
$ rpm -u *.rpm
```

Настоятельно рекомендуется сохранить резервную копию базы данных перед началом обновления. Подробные инструкции по выполнению обновления доступны на сайте PostgreSQL.

Резервное копирование и восстановление базы данных подробно рассматриваются в главе 11.

Состав дистрибутива PostgreSQL

К сожалению, во время написания этой книги стандарт на установку PostgreSQL еще не был выработан. Такой стандарт в большинстве случаев стал бы благом, хотя иногда и мог бы создавать некоторые трудности.

В состав PostgreSQL входят ряд приложений, утилит и каталогов с данными. Его главное приложение (`postmaster` или `postgres`) содержит

серверный код, отвечающий за обслуживание запросов на доступ к данным от клиентов. Утилиты, такие как `pg_ctl`, применяются для управления главным серверным процессом, который должен выполняться постоянно, обеспечивая работу сервера. В каталоге `data` находятся все файлы, необходимые базе данных. В них хранятся не только таблицы и записи, но и системные параметры.

По умолчанию устанавливаются все компоненты PostgreSQL, перечисленные в табл. 3.2, каждый в соответствующем подкаталоге. В качестве общего каталога (и каталога по умолчанию при установке из исходного кода) обычно используется `/usr/local/pgsql`.

Главный каталог PostgreSQL включает в себя следующие подкаталоги:

Таблица 3.2. Компоненты PostgreSQL

Каталог	Описание
`bin`	Приложения и утилиты, например `pg_ctl` и `postmaster`
`data`	Файлы базы данных
`doc`	Документация в формате HTML
`include`	Заголовочные файлы для разработки приложений PostgreSQL
`lib`	Библиотеки для разработки приложений PostgreSQL
`man`	Руководство по PostgreSQL
`share`	Примеры файлов конфигурации

Недостаток работы с единственным каталогом в том, что неизменные файлы программ и изменяющиеся данные хранятся совместно, а это не всегда удобно.

Файлы, используемые PostgreSQL, можно разделить на несколько категорий. Файлы программ не могут и не должны изменяться во время работы. Файлы журналов, создаваемые сервером, содержат информацию о доступе к базе данных и могут быть очень полезны в решении проблем. Они растут по мере того, как в них добавляются новые записи. Файлы данных – это сердце системы, в них хранится вся информация всех баз данных.

Из соображений эффективности и удобства администрирования файлы разных типов можно поместить в разные каталоги. Гибкость PostgreSQL позволяет хранить файлы программ, системных журналов и данных в различных местах, а дистрибутивы Linux позволяют извлечь из этой гибкости максимум выгоды.

Например, в SuSE 7.0 программы PostgreSQL размещаются вместе с остальными приложениями в `/usr/bin`, для журнальных файлов отводится `/var/log/postgresql`, а данные хранятся в `/var/lib/pgsql/data`. Благодаря этому можно легко разделить процедуры резервного копирования наиболее важных файлов данных и менее критичных журнальных файлов и т. п.

В других дистрибутивах могут применяться собственные схемы размещения файлов. Для просмотра списка файлов, установленных определенным пакетом, можно использовать утилиту RPM, запустив ее с такими параметрами:

```
$ rpm -q -l postgres
/usr/bin/createdb
...
/usr/share/man/manl/vacuum.1.gz
$
```

Чтобы просмотреть все установленные файлы, придется запустить `rpm` для каждого из пакетов, входящих в PostgreSQL:

```
$ rpm -q -l pg_serv
/sbin/conf.d/SuSEconfig.Postgres
...
/var/lib/pgsql/data/pg_options
$
```

В разных дистрибутивах названия пакетов могут слегка отличаться, в данном случае серверному пакету дано имя `pg_serv`. Вместо `rpm` можно использовать графический менеджер пакетов, например `kpackage` (рис. 3.2), поставляемый в комплекте с оболочкой KDE:

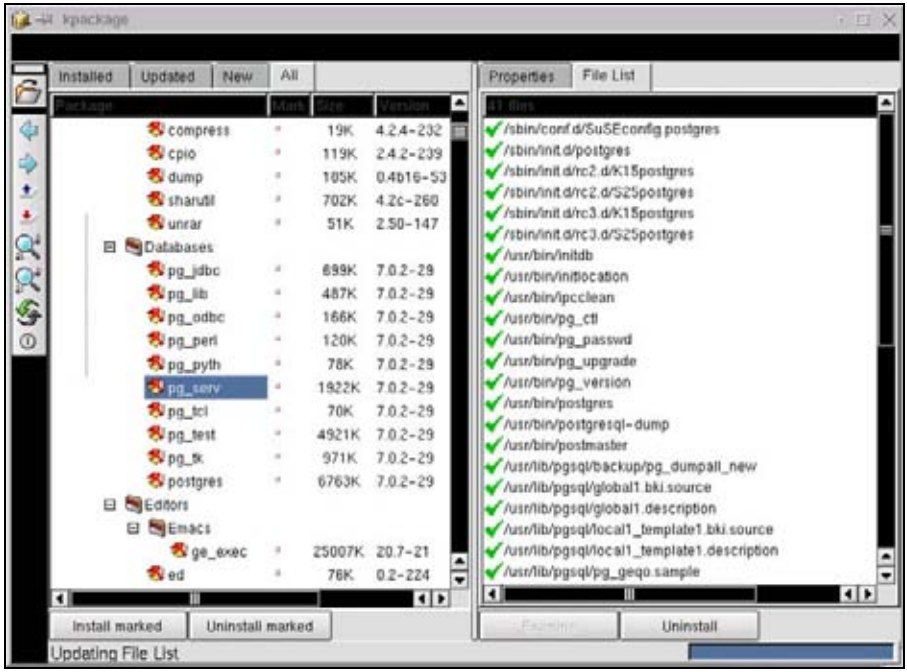


Рис. 3.2. Графический менеджер пакетов kpackage

Недостаток установки из дистрибутива Linux в том то, что не всегда очевидно, где какие файлы находятся. Так что тому, кто захочет обновить версию, может понадобиться настойчивость для того, чтобы разобраться в инсталляции.

Иногда бывает удобнее установить PostgreSQL из исходного кода. Тот, кто не планирует это делать, может пропустить следующий раздел и перейти к разделу «Запуск PostgreSQL».

Установка PostgreSQL из исходного кода

В то время как RPM пакеты предназначены для установки PostgreSQL в системах семейства Linux, исходный код позволяет собрать и установить PostgreSQL практически на любой UNIX-совместимой системе.

Исходный код последней версии, а часто и следующей бета-версии PostgreSQL можно получить на сайте *http://www.postgresql.org*. Если только вы не планируете находиться на переднем крае разработок, вполне можете ограничиться последней стабильной версией. На CD также записаны несколько предыдущих версий. Информацию о последних версиях и их доступности на CD можно получить на сайте PostgreSQL.

Исходный код распространяется в двух формах. Полный комплект исходного кода находится в файле с именем вроде

```
postgresql-7.1.2.tar.gz
```

Во время написания книги этот файл имел размер около 7,5 Мбайт. Кроме того, для облегчения загрузки исходный код упаковывается в несколько файлов меньшего размера:

```
postgresql-7.1.2.base.tar.gz
postgresql-7.1.2.docs.tar.gz
postgresql-7.1.2.opt.tar.gz
postgresql-7.1.2.test.tar.gz
```

Разумеется, эти имена тоже меняются в соответствии с номером текущей версии.

Компиляция PostgreSQL выполняется очень просто. Если вы уже компилировали продукты с открытым исходным кодом, то проблем не должно возникнуть. Впрочем, их не должно возникнуть, даже если это ваша первая попытка компиляции такого продукта. Для выполнения компиляции понадобится система Linux или UNIX с полностью установленной средой разработки, которая включает в себя компилятор C и GNU-версию утилиты `make`, необходимую для сборки системы.

Дистрибутивы Linux обычно поставляются со средой разработки, включающей GNU-инструментарий от фонда свободно распространяе-

мых программ FSF. Сюда входит превосходный компилятор GNU C (gcc), являющийся стандартным в Linux. Программы GNU доступны и для большинства других UNIX-платформ, поэтому мы рекомендуем использовать их для компиляции PostgreSQL. Последнюю версию инструментария можно получить по адресу *http://www.gnu.org*. Сразу после установки среды разработки можно переходить к компиляции PostgreSQL.

Скопируйте архивный файл с исходным кодом в какой-либо подходящий для компиляции каталог. Этот каталог – еще не то место, куда будет установлен PostgreSQL, он будет другим. Подходящим выбором будет подкаталог вашего домашнего подкаталога, поскольку для компиляции не требуются права суперпользователя, они понадобятся только при установке. Распакуйте архив:

```
$ tar zxvf postgres-7.1.2.tar.gz
```

Авторы предпочитают хранить исходные тексты в специальном каталоге для продуктов с открытым кодом /usr/src, но подойдет любой каталог, где достаточно свободного места для выполнения компиляции (понадобится около 50 Мбайт).

В процессе разархивирования будет создан новый каталог с именем, соответствующим версии PostgreSQL. Перейдите в этот каталог:

```
$ cd postgres-7.1.2
```

Там вы обнаружите файл INSTALL, содержащий подробные инструкции по сборке – на тот случай, если изложенный ниже метод автоматической сборки не даст результата.

В процессе сборки используется скрипт конфигурации configure, позволяющий настроить параметры сборки в соответствии с имеющимся окружением. При запуске без параметров configure руководствуется значениями по умолчанию:

```
$ ./configure
creating cache ./config.cache
checking host system type... i686-pc-linux-gnu
checking which template to use... linux
...
$
```

Скрипт configure устанавливает значения переменных, управляющих процессом сборки PostgreSQL, с учетом типа платформы, особенностей компилятора и т. д. Нюансы использования скрипта configure с различными значениями переменных выходят за рамки этой книги. Описание всех параметров можно найти в руководстве man.

Место установки выбирается скриптом configure автоматически. По умолчанию основным рабочим каталогом PostgreSQL назначается /usr/local/pgsql, а приложения и данные располагаются в его подкаталогах.

Запуск `configure` с параметрами позволяет изменить расположение каталогов. Чаще всего подставляются параметры, описанные в табл. 3.3:

Таблица 3.3. Параметры configure

Параметр	Описание
`--prefix=PREFIX`	Устанавливать в каталог `PREFIX`. По умолчанию – `/usr/local/psql`
`--bindir=DIR`	Устанавливать программы в `DIR`. По умолчанию – `PREFIX/bin`
`--with-tcl`	Компилировать с поддержкой приложений Tcl, таких как pgAccess
`--with-perl`	Компилировать с поддержкой приложений Perl
`--with-python`	Компилировать модуль поддержки Python

С помощью ключа `--help` можно получить полный список настроечных параметров:

```
$ ./configure --help

Usage: configure [options] [host]
Options: [defaults in brackets after descriptions]
Configuration:
  --cache-file=FILE       cache test results in FILE
...
$
```

Нет необходимости определять место расположения базы данных и журнальных файлов на данном этапе. Эти параметры могут быть заданы при запуске серверного процесса после того, как он уже установлен.

Закончив конфигурирование, можно приступать к сборке с помощью программы `make`.

> ***В процессе сборки PostgreSQL участвует ряд сложных make-файлов, управляющих компиляцией. Поэтому рекомендуется использовать GNU-make. В Linux эта версия установлена по умолчанию, но на других UNIX-платформах может потребоваться отдельная установка GNU-make. Часто ей дают имя*** `gmake`***, для того чтобы отличать от программы*** `gmake`***, поставляемой вместе с операционной системой. В дальнейшем обсуждении под make понимается GNU-make.***

Компилируем программы:

```
$ make
...
All of PostgreSQL successfully made. Ready to install.
```

Если все настроено правильно, будут выведены множество сообщений о компиляции и завершающее сообщение об успешном окончании сборки.

После завершения программы make необходимо скопировать программы в их рабочий каталог. Для этого мы также используем make, но прежде надо получить права суперпользователя:

```
$ su
# make install
...
Thank you for choosing PostgreSQL, the most advanced open source database
engine.
# exit
$
```

Вот, собственно, и все. Программы, составляющие сервер базы данных PostgreSQL, установлены.

Такой же результат мы получили бы при установке из пакетов RPM. Настало время выяснить, как же запускается PostgreSQL.

Запуск PostgreSQL

Основной процесс PostgreSQL – postmaster – это особого вида программа. Она полностью отвечает за доступ к данным – от всех пользователей и ко всем базам данных. Пользователю должны быть доступны его собственные данные и недоступны данные других пользователей, если он соответствующим образом не авторизован. Для этого серверный процесс должен владеть всеми файлами данных, запрещать прямой доступ к ним для обычных пользователей и выдавать разрешение на доступ, проверяя права пользователя, пославшего запрос.

Для управления доступом к данным в PostgreSQL реализована концепция псевдопользователей. Пользователь по имени postgres создается с единственной целью – владеть файлами данных. Регистрация в системе под этим именем с целью получения неавторизованного доступа невозможна. Программа postmaster по этому имени организует доступ к файлам данных от имени других пользователей.

Таким образом, первым шагом к созданию работающей базы данных PostgreSQL будет создание пользователя postgres.

В разных системах процедура создания пользователя может выглядеть по-разному. Пользователи Linux могут (зарегистрировавшись как root) выполнить команду useradd:

```
# useradd postgres
```

В других UNIX-системах может потребоваться создание домашнего каталога, редактирование файлов конфигурации или запуск административных программ. О программах администрирования можно узнать в документации по операционной системе.

Теперь надо создать и произвести начальную инициализацию базы данных и запустить процесс postmaster.

Используем утилиту initdb для инициализации, указав ей место в файловой системе, где должны быть расположены файлы данных. Предварительно надо создать (из-под пользователя root) каталог и назначить пользователя postgres его владельцем:

```
# mkdir /usr/local/pgsql/data
# chown postgres /usr/local/pgsql/data
```

Здесь выбрано расположение базы данных по умолчанию. Как говорилось ранее, каталог может быть и другим.

Для того чтобы инициализировать базу данных, необходимо запустить соответствующие программы PostgreSQL от имени пользователя postgres, как это показано ниже.

Для того чтобы зарегистрироваться как postgres, надо сначала войти в систему как суперпользователь (root), а затем выполнить команду su:

```
$ su
# su - postgres
pg$
```

Теперь все запускаемые программы получают права пользователя postgres и имеют доступ к файлам базы данных PostgreSQL. Для наглядности приглашение оболочки, запущенной под пользователем postgres, обозначено как pg$.

> ***Не следует поддаваться искушению упростить процесс, запуская программы с правами пользователя*** root. ***Серверные процессы, запущенные с правами суперпользователя, могут ослабить защищенность системы. В случае сбоев в серверном процессе пользователь может получить возможность проникновения в систему по сети. По этой причине программа*** postmaster ***отказывается запускаться под пользователем*** root.

Инициализируем базу данных программой initdb:

```
pg$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
This database system will be initialized with the user name "Postgres".
This user will own all the data files and must also own the server process.
...
Success.
pg$
```

Если все пройдет хорошо, получим новую, совершенно пустую базу данных, расположенную в каталоге, который был указан параметром -D.

Теперь запустим собственно серверный процесс. Снова укажем параметр -D, чтобы сообщить программе postmaster, где находятся файлы базы данных. Для того чтобы пользователи могли получить доступ к данным по сети, следует указать параметр -i, разрешающий удаленный доступ:

```
pg$ /usr/local/pgsql/bin/postmaster -i -D /usr/local/pgsql/data >logfile 2>&1 &
```

По умолчанию PostgreSQL не разрешает удаленный доступ. Для того чтобы разрешить устанавливать соединения, нужно отредактировать файл конфигурации `pg_hba.conf`. Этот файл находится там же, где и файлы базы данных (в нашем примере – `/usr/local/pgsql/data`), и содержит записи, разрешающие или запрещающие удаленным пользователям соединяться с базой данных. Формат файла достаточно простой, а типовой файл, поставляемый с PostgreSQL, содержит множество комментариев, которые помогут добавить новые записи. Имеется возможность выдавать разрешения отдельным пользователям, компьютерам или группам компьютеров на доступ к отдельным базам данных.

Добавим запись, разрешающую любому компьютеру локальной сети устанавливать соединение с любой базой данных без аутентификации. Для того чтобы задать другую политику доступа, обратитесь к комментариям в файле конфигурации.

Добавим в конец файла `pg_hba.conf` строку такого вида:

```
host    all    192.168.0.0    255.255.0.0    trust
```

Это означает, что все компьютеры, IP-адрес которых начинается с 192.168, имеют доступ ко всем базам данных.

Мы также перенаправили вывод процесса в файл (`logfile` в домашнем каталоге пользователя `postgres`) и объединили стандартный вывод со стандартным выводом ошибок, с помощью конструкции `2>&1`. Разумеется, можно выбрать и другое расположение журнального файла, перенаправив в него вывод.

Теперь попробуем установить соединение с базой данных, чтобы убедиться, что она работает. Для выполнения простых задач администрирования, таких как создание пользователей, таблиц и баз данных, применяется утилита `psql`. Ниже в этой главе мы с ее помощью создадим и заполним пробную базу данных. А сейчас попробуем просто присоединиться к базе данных. Полученный ответ говорит о том, что процесс `postmaster` успешно запустился:

```
pg$ /usr/local/pgsql/bin/psql
psql: FATAL 1: Database "postgres" does not exist in the system catalog.
```

Пусть вас не смущает сообщение об ошибке: по умолчанию `psql` соединяется с базой данных на локальной машине и пытается открыть базу данных с тем же именем, что и у пользователя, запустившего программу. Попытка закончилась неудачно, т. к. мы не создавали базу данных с именем `postgres`. Однако такой ответ означает, что процесс `postmaster` выполняется и выдает диагностические сообщения.

Чтобы сообщить программе `psql`, с какой базой данных следует устанавливать соединение, надо указать параметр `-d`. Только что установленная система PostgreSQL содержит несколько баз данных, используемых в качестве основы для вновь создаваемых баз. Одна из них на-

зывается template1. При необходимости с этой базой данных можно установить соединение и выполнить какие-либо административные задачи.

Для того чтобы проверить работу в сети, установим соединение с другой машины, на которой также установлена PostgreSQL. Параметром -h определим компьютер (по имени или по IP-адресу) и соединимся с одной из системных баз данных (поскольку собственных баз мы еще не создали):

```
remote$ psql -h 192.168.0.66 -d template1
Welcome to psql, the PostgreSQL interactive terminal.

Type:    \copyright for distribution terms
       \h for help with SQL commands
       \? For help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit
template1=# \q

remote$
```

Последнее, что следует сделать – настроить автоматический запуск серверного процесса postmaster при каждом рестарте машины.

Собственно, надо лишь обеспечить запуск программы postmaster во время загрузки. Здесь, опять-таки, нет единых стандартов для всех версий Linux и UNIX. Детальное описание можно найти в документации по системе.

Если PostgreSQL был установлен из дистрибутива Linux, то, скорее всего, процедура запуска уже настроена RPM-пакетом. В системе SuSE Linux при переходе в многопользовательский режим PostgreSQL запускается автоматически с помощью командного файла /etc/rc.d/init.d, вызывающего программу postgres.

Простейший способ создать программу запуска самостоятельно состоит в том, чтобы написать командный файл, запускающий postmaster с нужными параметрами, и вызывать его из файлов автозапуска, расположенных в /etc/rc.d. Не забывайте, что процесс postmaster должен запускаться пользователем postgres.

Здесь представлен сценарий, запускающий PostgreSQL, установленный из исходных текстов, с параметрами по умолчанию:

```
#!/bin/sh

# Script to start and stop PostgreSQL

SERVER=/usr/local/pgsql/bin/postmaster
PGCTL=/usr/local/pgsql/bin/pg_ctl
PGDATA=/usr/local/pgsql/data
```

```
OPTIONS=-i
LOGFILE=/usr/local/pgsql/data/postmaster.log

case "$1" in
    start)
        echo -n "Starting PostgreSQL..."
       su -l postgres -c "nohup $SERVER $OPTIONS -D $PGDATA >$LOGFILE 2>&1 &"
        ;;
    stop)
        echo -n "Stopping PostgreSQL..."
        su -l postgres -c "$PGCTL -D $PGDATA stop"
        ;;
    *)
        echo "Usage: $0 {start|stop}"
        exit 1
        ;;
esac
exit 0
```

В Debian Linux может потребоваться замена `su -l` ***на*** `su -`.

Поместите приведенную программу в исполняемый файл и назовите его, к примеру, MyPostgreSQL. Выполните команду `chmod`, чтобы сделать файл исполняемым:

```
# chmod a+rx MyPostgreSQL
```

Теперь следует сделать так, чтобы файл вызывался, когда PostgreSQL запускается и останавливается при запуске и остановке сервера:

```
MyPostgreSQL start
MyPostgreSQL stop
```

В системах, использующих команды инициализации в стиле System V (а это большинство дистрибутивов Linux), достаточно поместить команду в соответствующее место. Например, в SuSE Linux следует поместить файл в `/etc/rc.d/init.d/MyPostgreSQL` и для автоматического запуска и остановки PostgreSQL при переходе системы в многопользовательский режим и обратно создать следующие символические ссылки на него:

```
/etc/rc.d/rc2.d/S25MyPostgreSQL
/etc/rc.d/rc2.d/K25MyPostgreSQL
/etc/rc.d/rc3.d/S25MyPostgreSQL
/etc/rc.d/rc3.d/K25MyPostgreSQL
```

Обратитесь к документации по операционной системе за дополнительными сведениями о командах запуска.

Пришло время создать базу данных.

Создание базы данных

Позже мы вернемся к администрированию баз данных и рассмотрим его более подробно, а сейчас, когда у нас уже есть работающая база данных, рассмотрим эту тему вкратце. Создадим простую базу данных, которую назовем bpsimple, и будем использовать ее в учебных целях. В дальнейшем в примерах, демонстрирующих возможности PostgreSQL, будем иметь в виду именно ее.

Прежде чем начать, убедимся, что PostgreSQL запущен, для этого найдем postmaster в списке работающих процессов:

```
$ ps -el | grep post
```

Если процесс с именем postmaster запущен (имя на дисплее может быть сокращено), значит, сервер PostgreSQL работает.

Каждый пользователь PostgreSQL может создавать собственные базы данных и управлять доступом к хранимым в них данным. Но прежде чем создавать базы данных, необходимо создать в системе записи о пользователях PostgreSQL. Для этого запустим утилиту createuser.

С помощью команды su (из-под пользователя root) зарегистрируемся как пользователь postgres. Затем запустим утилиту createuser, регистрирующую пользователя. Пользователю с указанным именем будет разрешена работа с PostgreSQL. Зарегистрируем пользователя neil (уже созданного в UNIX/Linux):

```
$ su
# su - postgres
pg$ /usr/local/pgsql/bin/createuser neil
Shall the new user be able to create databases? (y/n) y
Shall the new user be able to create new users (y/n) n
CREATE USER
pg$
```

Здесь пользователю neil предоставлено право создавать новые базы данных, но не дано право создавать других пользователей.

Теперь, когда зарегистрирован пользователь PostgreSQL с такими правами, можно создать базу данных. Вернитесь в собственную сессию (не в сессию root) и введите:

```
$ /usr/local/pgsql/bin/createdb bpsimple
CREATE DATABASE
$
```

Теперь можно установить соединение (локальное) с сервером, используя интерактивный терминал psql:

```
$ /usr/local/pgsql/bin/psql -d bpsimple
Welcome to psql, the PostgreSQL interactive terminal.
...
bpsimple=#
```

Вы зарегистрировались в PostgreSQL и можете выполнять некоторые команды. Для возврата в оболочку выполните команду /q.

Создание таблиц

Таблицы в базе bpsimple можно создавать, вводя команды в ответ на приглашение psql, но гораздо проще будет загрузить файл с набором сценариев с сайта издательства Wrox, распаковать его и запустить командой \i <имяфайла>, которая выполняет команды, считывая их из файла. Команды находятся в обычном текстовом файле, так что при желании их можно редактировать в любом текстовом редакторе:

```
bpsimple=# \i create_tables.sql
CREATE TABLE
...
bpsimple=#
```

Можно рекомендовать написание командного файла для создания схемы (то есть таблиц, индексов, процедур). В этом случае при необходимости повторного создания базы данных можно будет запустить готовый сценарий. То же относится и к изменениям в схеме.

Файл create_tables.sql, входящий в набор, содержит SQL-операторы создания таблиц:

```
create table customer
(
    customer_id             serial                      ,
    title                   char(4)                     ,
    fname                   varchar(32)                 ,
    lname                   varchar(32)          not null,
    addressline             varchar(64)                 ,
    town                    varchar(32)                 ,
    zipcode                 char(10)             not null,
    phone                   varchar(16)                 ,
    CONSTRAINT              customer_pk PRIMARY KEY(customer_id)
);


create table item
(
    item_id                 serial                      ,
    description             varchar(64)          not null,
    cost_price              numeric(7,2)                ,
    sell_price              numeric(7,2)                ,
    CONSTRAINT              item_pk PRIMARY KEY(item_id)
);


create table orderinfo
(
```

```
    orderinfo_id            serial                      ,
    customer_id             integer             not null,
    date_placed             date                not null,
    date_shipped            date                        ,
    shipping                numeric(7,2)                ,
    CONSTRAINT              orderinfo_pk PRIMARY KEY(orderinfo_id)
);

create table stock
(
    item_id                 integer             not null,
    quantity                integer             not null,
    CONSTRAINT              stock_pk PRIMARY KEY(item_id)
);

create table orderline
(
    orderinfo_id            integer             not null,
    item_id                 integer             not null,
    quantity                integer             not null,
    CONSTRAINT              orderline_pk PRIMARY KEY(orderinfo_id, item_id)
);

create table barcode
(
    barcode_ean             char(13)            not null,
    item_id                 integer             not null,
    CONSTRAINT              barcode_pk PRIMARY KEY(barcode_ean)
);
```

Удаление таблиц

Если в какой-то момент мы решим удалить все таблицы и начать все сначала, это можно будет сделать. Необходимый для этого набор команд находится в файле `drop_tables.sql`:

```
drop table barcode;

drop table orderline;

drop table stock;

drop table orderinfo;

drop table item;

drop table customer;
```

```
drop sequence customer_customer_id_seq;

drop sequence item_item_id_seq;

drop sequence orderinfo_orderinfo_id_seq;
```

Будьте осторожны: удаляя таблицы, вы удаляете и все хранящиеся в них данные!

Если этот сценарий запускается после создания таблиц, то запустите повторно `create_tables.sql` перед тем, как заполнять таблицы данными.

Заполнение таблиц

Последний по порядку, но не по важности шаг – заполнение таблиц, то есть занесение в них данных.

Фигурирующие в примерах данные имеются в файле `pop_tablename.sql`. Вы, конечно, можете взять и собственные – с учетом того, что и результаты будут отличаться от приведенных в книге. Поэтому, если вы еще не стали специалистом, наверное, лучше использовать наши данные.

Наличие переносов строк обусловлено лишь форматом печатной страницы, команды можно вводить по одной на строке. Не забывайте о завершающей точке с запятой, она сообщает `psql`, где заканчивается SQL-команда.

Таблица customer

```
insert into customer(title, fname, lname, addressline, town, zipcode, phone)
values('Miss','Jenny','Stones','27 Rowan Avenue','Hightown','NT2 1AQ','023
9876');
insert into customer(title, fname, lname, addressline, town, zipcode, phone)
values('Mr','Andrew','Stones','52 The Willows','Lowtown','LT5 7RA','876
3527');
insert into customer(title, fname, lname, addressline, town, zipcode, phone)
values('Miss','Alex','Matthew','4 The Street','Nicetown','NT2 2TX','010
4567');
insert into customer(title, fname, lname, addressline, town, zipcode, phone)
values('Mr','Adrian','Matthew','The Barn','Yuleville','YV67 2WR','487
3871');
insert into customer(title, fname, lname, addressline, town, zipcode, phone)
values('Mr','Simon','Cozens','7 Shady Lane','Oahenham','OA3 6QW','514
5926');
insert into customer(title, fname, lname, addressline, town, zipcode, phone)
values('Mr','Neil','Matthew','5 Pasture Lane','Nicetown','NT3 7RT','267
1232');
```

```
insert into customer(title, fname, lname, addressline, town, zipcode, phone)
values('Mr','Richard','Stones','34 Holly Way','Bingham','BG4 2WE','342 5982');
insert into customer(title, fname, lname, addressline, town, zipcode, phone)
values('Mrs','Ann','Stones','34 Holly Way','Bingham','BG4 2WE','342 5982');
insert into customer(title, fname, lname, addressline, town, zipcode, phone)
values('Mrs','Christine','Hickman','36 Queen Street','Histon','HT3 5EM','342
5432');
insert into customer(title, fname, lname, addressline, town, zipcode, phone)
values('Mr','Mike','Howard','86 Dysart Street','Tibsville','TB3 7FG','505
5482');
insert into customer(title, fname, lname, addressline, town, zipcode, phone)
values('Mr','Dave','Jones','54 Vale Rise','Bingham','BG3 8GD','342 8264');
insert into customer(title, fname, lname, addressline, town, zipcode, phone)
values('Mr','Richard','Neill','42 Thached way','Winersby','WB3 6GQ','505
6482');
insert into customer(title, fname, lname, addressline, town, zipcode, phone)
values('Mrs','Laura','Hendy','73 Margeritta Way','Oxbridge','OX2 3HX','821
2335');
insert into customer(title, fname, lname, addressline, town, zipcode, phone)
values('Mr','Bill','O\'Neill','2 Beamer Street','Welltown','WT3 8GM','435
1234');
insert into customer(title, fname, lname, addressline, town, zipcode, phone)
values('Mr','David','Hudson','4 The Square','Milltown','MT2 6RT','961 4526');
```

Таблица item

```
insert into item, (description, cost_price, sell_price) values('Wood Puzzle',
15.23, 21.95);
insert into item(description, cost_price, sell_price) values('Rubic Cube',
7.45, 11.49);
insert into item(description, cost_price, sell_price) values('Linux CD',
1.99, 2.49);
insert into item(description, cost_price, sell_price) values('Tissues',
2.11, 3.99);
insert into item(description, cost_price, sell_price) values('Picture
Frame', 7.54, 9.95);
insert into item(description, cost_price, sell_price) values('Fan Small',
9.23, 15.75);
insert into item(description, cost_price, sell_price) values('Fan Large',
13.36, 19.95);
insert into item(description, cost_price, sell_price) values('Toothbrush',
0.75, 1.45);
insert into item(description, cost_price, sell_price) values('Roman Coin',
2.34, 2.45);
insert into item(description, cost_price, sell_price) values('Carrier Bag',
0.01, 0.0);
insert into item(description, cost_price, sell_price) values('Speakers',
19.73, 25.32);
```

Таблица barcode

```
insert into barcode(barcode_ean, item_id) values('6241527836173', 1);
insert into barcode(barcode_ean, item_id) values('6241574635234', 2);
insert into barcode(barcode_ean, item_id) values('6264537836173', 3);
insert into barcode(barcode_ean, item_id) values('6241527746363', 3);
insert into barcode(barcode_ean, item_id) values('7465743843764', 4);
insert into barcode(barcode_ean, item_id) values('3453458677628', 5);
insert into barcode(barcode_ean, item_id) values('6434564564544', 6);
insert into barcode(barcode_ean, item_id) values('8476736836876', 7);
insert into barcode(barcode_ean, item_id) values('6241234586487', 8);
insert into barcode(barcode_ean, item_id) values('9473625532534', 8);
insert into barcode(barcode_ean, item_id) values('9473627464543', 8);
insert into barcode(barcode_ean, item_id) values('4587263646878', 9);
insert into barcode(barcode_ean, item_id) values('9879879837489', 11);
insert into barcode(barcode_ean, item_id) values('2239872376872', 11);
```

Таблица orderinfo

```
insert into orderinfo(customer_id, date_placed, date_shipped, shipping)
values(3,'03-13-2000','03-17-2000', 2.99);
insert into orderinfo(customer_id, date_placed, date_shipped, shipping)
values(8,'06-23-2000','06-24-2000', 0.00);
insert into orderinfo(customer_id, date_placed, date_shipped, shipping)
values(15,'09-02-2000','09-12-2000', 3.99);
insert into orderinfo(customer_id, date_placed, date_shipped, shipping)
values(13,'09-03-2000','09-10-2000', 2.99);
insert into orderinfo(customer_id, date_placed, date_shipped, shipping)
values(8,'07-21-2000','07-24-2000', 0.00);
```

Таблица orderline

```
insert into orderline(orderinfo_id, item_id, quantity) values(1, 4, 1);
insert into orderline(orderinfo_id, item_id, quantity) values(1, 7, 1);
insert into orderline(orderinfo_id, item_id, quantity) values(1, 9, 1);
insert into orderline(orderinfo_id, item_id, quantity) values(2, 1, 1);
insert into orderline(orderinfo_id, item_id, quantity) values(2, 10, 1);
insert into orderline(orderinfo_id, item_id, quantity) values(2, 7, 2);
insert into orderline(orderinfo_id, item_id, quantity) values(2, 4, 2);
insert into orderline(orderinfo_id, item_id, quantity) values(3, 2, 1);
insert into orderline(orderinfo_id, item_id, quantity) values(3, 1, 1);
insert into orderline(orderinfo_id, item_id, quantity) values(4, 5, 2);
insert into orderline(orderinfo_id, item_id, quantity) values(5, 1, 1);
insert into orderline(orderinfo_id, item_id, quantity) values(5, 3, 1);
```

Таблица stock

```
insert into stock(item_id, quantity) values(1,12);
insert into stock(item_id, quantity) values(2,2);
insert into stock(item_id, quantity) values(4,8);
```

```
insert into stock(item_id, quantity) values(5,3);
insert into stock(item_id, quantity) values(7,8);
insert into stock(item_id, quantity) values(8,18);
insert into stock(item_id, quantity) values(10,1);
```

Теперь, запустив сервер и создав базу данных, создав и заполнив таблицы, можно продолжить изучение PostgreSQL.

Остановка PostgreSQL

Очень важно, чтобы серверный процесс PostgreSQL был правильно остановлен. Только в этом случае он сможет завершить запись в базу данных и освободить занятую им разделяемую память.

Для того чтобы корректно остановить базу данных, зарегистрируйтесь как `postgres` или `root` и запустите утилиту `pg_ctl`:

```
# /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data stop
```

Конечно, если созданы сценарии для запуска, то можно использовать команду:

```
# /etc/rc.d/init.d/MyPostgreSQL stop
```

Эти сценарии выполняют также корректное закрытие базы данных при остановке и перезагрузке машины.

Ресурсы

Дополнительную информацию об утилитах `psql`, `createuser` и `createdb` можно получить из справочных страниц, воспользовавшись командой `man`, и из другой документации, поставляемой с PostgreSQL. Для более удобной работы с PostgreSQL имеет смысл добавить каталоги с его приложениями и руководством в соответствующие пути поиска. В стандартной UNIX-оболочке для этого следует добавить следующие команды в свой стартовый файл в домашнем каталоге (`.profile` или `.bashrc`):

```
PATH=$PATH:/usr/local/pgsql/bin
MANPATH=$MANPATH:/usr/local/pgsql/man
export PATH MANPATH
```

Как уже упоминалось, исходные тексты текущей и последней из тестовых версий PostgreSQL можно получить на сайте *http://www.postgresql.org*. Другие источники сведений о PostgreSQL приведены в главе 18.

Установка PostgreSQL в Windows

Начнем этот раздел с хорошей новости для пользователей Windows. Хотя PostgreSQL и был разработан для UNIX-подобных платформ, при

его написании учитывались соображения переносимости. Сначала появилась возможность писать клиентские приложения PostgreSQL для Windows, а начиная с версии 7.1 сервер PostgreSQL может быть скомпилирован, установлен и запущен в системе Microsoft Windows NT4 и Windows 2000. Для этого придется установить дополнительное программное обеспечение, реализующее некоторые возможности UNIX в Windows.

Порядок установки во многом совпадает с тем, который применялся при установке на Linux и UNIX из исходных текстов. Отличия заключаются в том, что придется установить UNIX-подобную среду разработки в Windows и выполнить пару нетривиальных действий, чтобы процесс `postmaster` запускался автоматически. Тем не менее при необходимости это может быть сделано. В действительности, для целей тестирования новой базы данных и решения проблем с сетевыми соединениями бывает весьма полезно иметь такую утилиту PostgreSQL, как `psql`, на той же машине, на которой установлено клиентское приложение, даже если сам сервер не должен работать в Windows.

Cygwin – UNIX-среда для Windows

В помощь тем, кто хотел бы расширить возможности Microsoft Windows, компания Cygnus Solutions (теперь ставшая частью Red Hat) разработала набор библиотек, эмулирующих некоторые UNIX API в среде Windows NT и 2000. Эти DLL позволяют программам, написанным для UNIX или Linux, выполняться в Windows. Имеется ряд ограничений, в особенности в части прав доступа и пользовательских привилегий, связанных с различием архитектур операционных систем, но, тем не менее, это весьма полезное дополнение к открытому программному обеспечению.

Пакет называется Cygwin и может быть свободно получен с сайта *http://www.cygwin.com*. В этом разделе будет показано, как установить Cygwin на платформе Windows и использовать его для компиляции исходных текстов PostgreSQL.

Прежде всего, зайдите на сайт *http://www.cygwin.com* и пойдите по ссылке «Install Cygwin now» (рис. 3.3).

Рис. 3.3. Установка Cgwin

Это ссылка на программу установки, которую можно загрузить и выполнить на локальной машине, а можно запустить прямо с сайта. Но мы рекомендуем загрузить ее на Windows-машину и запускать с локального жесткого диска.

Полная дистрибутивная копия Cygwin имеет размер более 40 Мбайт.

Запустите программу `setup.exe`, чтобы начать установку (рис. 3.4).

Программа содержит пошаговые инструкции по загрузке и установке многочисленных пакетов, входящих в набор инструментов Cygwin.

На следующем шаге надо выбрать стратегию установки. Можно загружать файлы в процессе установки. Можно сначала только загрузить файлы, а устанавливать позже (рис. 3.5). Мы рекомендуем второй способ, в этом случае файлы можно будет использовать для повторной установки или для установки на другой машине.

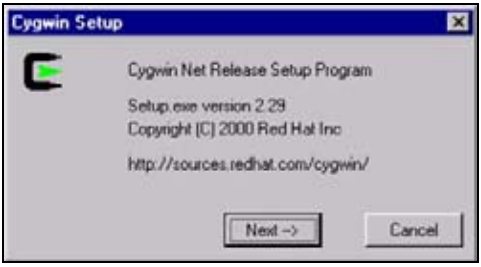


Рис. 3.4. Начало установки

Рис. 3.5. Выбор способа установки

После загрузки всех файлов из Интернета следует еще раз запустить программу `setup.exe` и выбрать пункт Install from Local Directory, указав местоположение загруженных файлов. Программа предлагает выбрать каталог для загружаемых файлов и задает вопрос о типе установленного соединения (рис. 3.6). Если установлен броузер Internet Explorer 5 и выше, то программа `setup.exe` может руководствоваться его настройками.

Рис. 3.6. Выбор типа соединения

Теперь надо выбрать сайт, с которого будет выполняться загрузка. Имеется ряд сайтов, расположенных по всему миру (рис. 3.7), которые хранят копии Cygwin, что позволяет избежать перегрузок на главном сайте. Рекомендуется выбирать сайт, географически наиболее близкий.

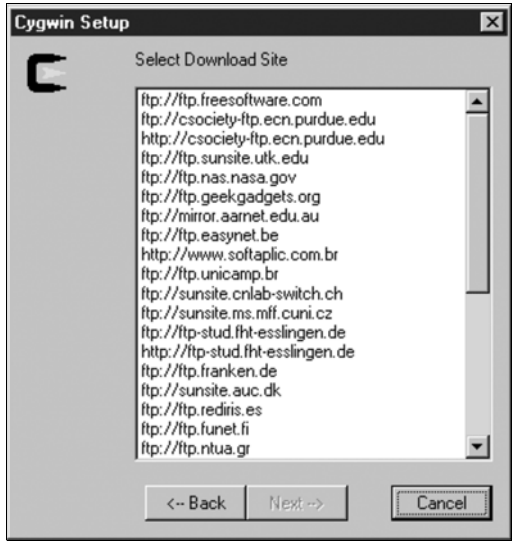


Рис. 3.7. Выбор сайта

Следующий шаг заключается в выборе пакетов для загрузки и установки из большого числа имеющихся в Cygwin (рис. 3.8). По умолчанию копируются только откомпилированные пакеты, хотя исходные тексты тоже доступны. Для компиляции PostgreSQL вполне подойдет выбор по умолчанию.

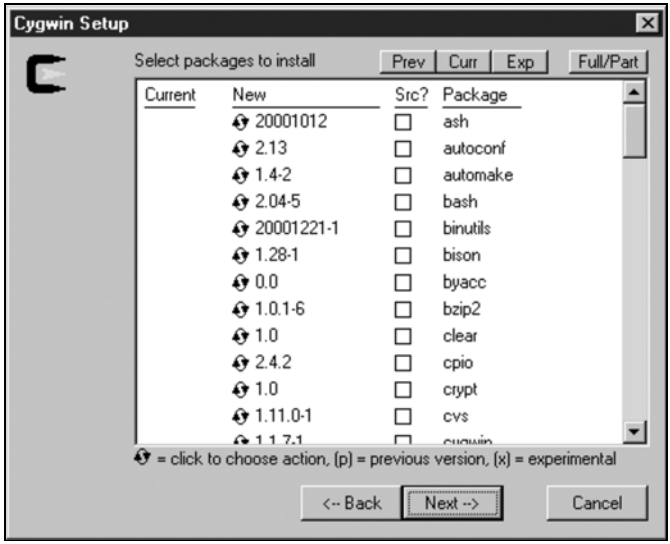


Рис. 3.8. Выбор пакетов

Если вы уверены, что какой-либо из пакетов вам не понадобится, можете пропустить его, установив признак `skip`. Если доступны несколь-

ко версий пакета, можно выбрать любую из них. Как правило, лучше выбирать более позднюю версию. Теперь программа `setup.exe` может начать загрузку (рис. 3.9):

Рис. 3.9. Загрузка началась...

По окончании загрузки можно запустить `setup.exe` еще раз и выбрать установку из локального каталога, в котором находятся загруженные файлы (рис. 3.10):

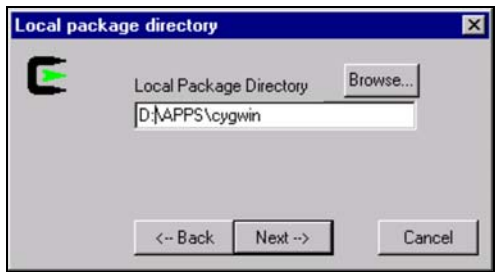


Рис. 3.10. Локальный каталог

Теперь пришло время выбрать место для установки Cygwin. В процессе установки Cygwin создает дерево каталогов, аналогичное традиционно используемому в UNIX и Linux. В нем присутствует корневой каталог с подкаталогами `bin`, `lib`, `usr` и т. д. Это дерево будет помещено в указанный каталог файловой системы NT/2000 (рис. 3.11):

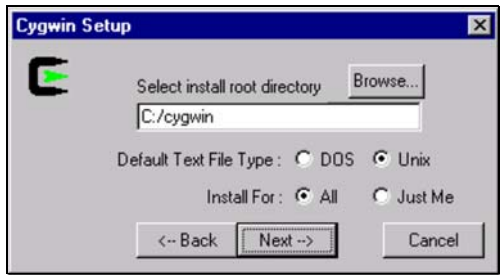


Рис. 3.11. Выбор каталога установки

Мы рекомендуем выбрать значение UNIX для типа текстовых файлов по умолчанию (Default Text File Type), поскольку в этом случае фай-

лы, создаваемые одними программами Cygwin, наверняка смогут быть прочитаны другими его программами.

При необходимости редактирования файлов Cygwin в приложениях Windows предпочтительно использовать редакторы, распознающие различные форматы текстовых файлов и умеющие с ними правильно обращаться. Хорошим выбором будет файловый редактор программиста (Programmer's File Editor, PFE), доступный на многих сайтах со свободно распространяемыми программами. Домашняя страница PFE находится по адресу *http://www.lancs.ac.uk/people/cpaap/pfe*. Имеются также версии замечательных UNIX-редакторов `vi` и `emacs` для Cygwin.

Если выбрать вариант установки Install For All (Устанавливать для всех), группа программ Cygwin будет доступна всем пользователям Windows, в противном случае – только вам. Рекомендуем выбрать установку для всех, чтобы пользователь `postgres` мог запускать программы Cygwin.

И наконец, надо выбрать программы, которые будут установлены; по умолчанию устанавливаются все загруженные пакеты.

По окончании установки на рабочем столе появятся значок и программная группа Cygnus (рис. 3.12), запускающая UNIX-оболочку в среде Windows:

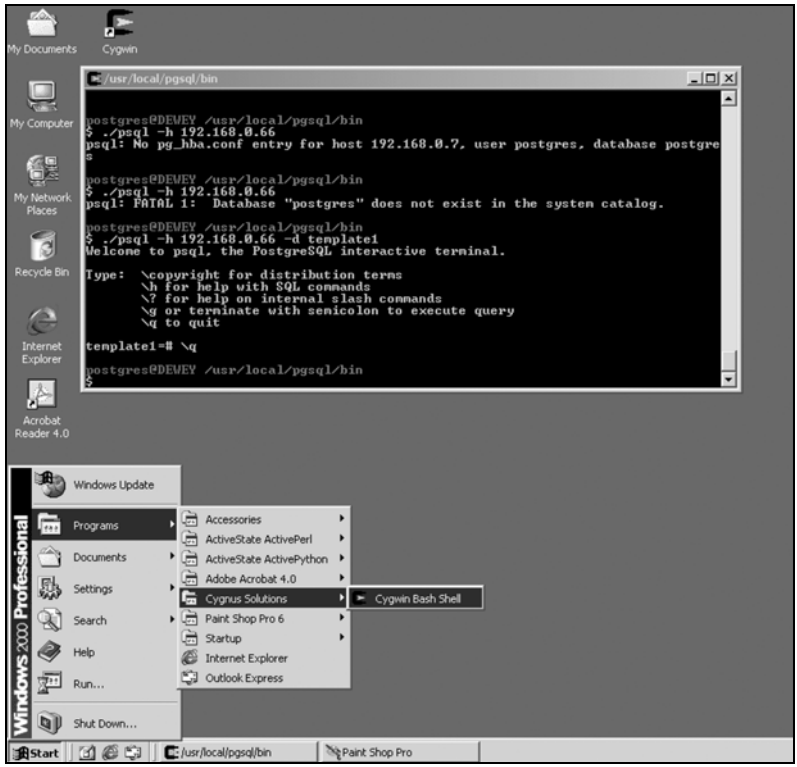


Рис. 3.12. Обновленный рабочий стол

Среди установленных программ Cygwin имеются компилятор GNU C, командная оболочка Bash и практически все необходимые средства для компиляции исходных текстов PostgreSQL.

Службы IPC для Windows

Еще один компонент программного обеспечения, необходимого для работы PostgreSQL под Windows, – это менеджер IPC, предназначенный для управления разделяемой памятью и семафорами. Соответствующий пакет, не входящий по умолчанию в Cygwin, называется CygIPC и доступен по адресу *http://www.neuro.gatech.edu/users/cwilson/cygutils/V1.1/cygipc*.

Установка CygIPC не вызывает проблем. Просто загрузите пакет и распакуйте его в корневом каталоге Cygwin:

```
$ cd /
$ tar zxvf cygipc-1.09-2.tar.gz
```

PostgreSQL для Cygwin

Начиная с версии 2.29 в Cygwin в качестве дополнительного пакета входит дистрибутив PostgreSQL, который может быть проинсталлирован программой установки Cygwin. Программы PostgreSQL помещаются в каталог `/usr/bin`, библиотеки – в `/usr/lib`, а разделяемые файлы – в `/usr/share`.

Компиляция PostgreSQL в Windows

Последовательность шагов по компиляции PostgreSQL в Windows NT/2000 практически та же, что и в Linux и UNIX, поэтому рассмотрим ее вкратце.

Скопируйте архив с исходными текстами (`postgresql-7.1.2.tar.gz`) в подкаталог корневого каталога Cygwin, например `/usr/src`.

> ***Корневой каталог Cygwin доступен из программы Windows Explorer как каталог, в который установлен Cygwin.***

Распакуйте исходные тексты и запустите `configure` и `make`, как и в случае с UNIX:

```
$ tar zxvf postgresql-7.1.2.tar.gz
$ cd postgres-7.1.2
$ ./configure
$ make
$ make install
```

Теперь PostgreSQL установлена в каталоге `/usr/local/pgsql`.

Конфигурирование PostgreSQL в Windows

Так же как в UNIX и Linux, серверный процесс PostgreSQL `postmaster` должен выполняться под специальным пользователем. Поэтому начнем с того, что создадим пользователя `postgres`.

Запустите приложение «Пользователи и пароли» панели управления, чтобы создать пользователя `postgres` как члена группы администраторов. Назначьте пароль обычным образом. Пароль необходим для запуска служб Windows, поэтому советуем установить для него неограниченный срок действия.

Теперь вы можете зарегистрироваться как пользователь `postgres` и запустить в Cygwin сессию `bash`.

Прежде чем запускать программу `postmaster`, необходимо запустить службу, реализующую IPC UNIX-типа. Это делается командой

```
$ /usr/local/bin/ipc-daemon.exe &
```

Теперь можно инициализировать базу данных – так же, как в среде UNIX и Linux:

```
pg$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

Чтобы разрешить удаленный доступ, необходимо перед запуском программы `postmaster` отредактировать файл `pg_hba.conf`, добавив в конце строку:

```
host    all    192.168.0.0    255.255.0.0    trust
```

Это означает, что все компьютеры, IP-адреса которых начинаются с 192.168, могут получить доступ к базе данных.

Запустите серверный процесс той же командой, что и раньше:

```
pg$ /usr/local/pgsql/bin/postmaster -i -D /usr/local/pgsql/data
>logfile 2>&1 &
```

Как и раньше, можно создавать пользователей и базы данных:

```
$ /usr/local/pgsql/bin/createuser neil
$ /usr/local/pgsql/bin/createdb test
```

Используя программу `psql` локально или удаленно, можно проверить, запущена ли база данных. На снимках экрана показано, как PostgreSQL выполняется под Windows 2000, а доступ осуществляется с Linux-машины (рис. 3.13 и 3.14).

То же самое можно сделать в Windows, запустив в Cygwin вторую сессию `bash` и выполнив команду

```
$ /usr/local/pgsql/bin/psql -d test
```

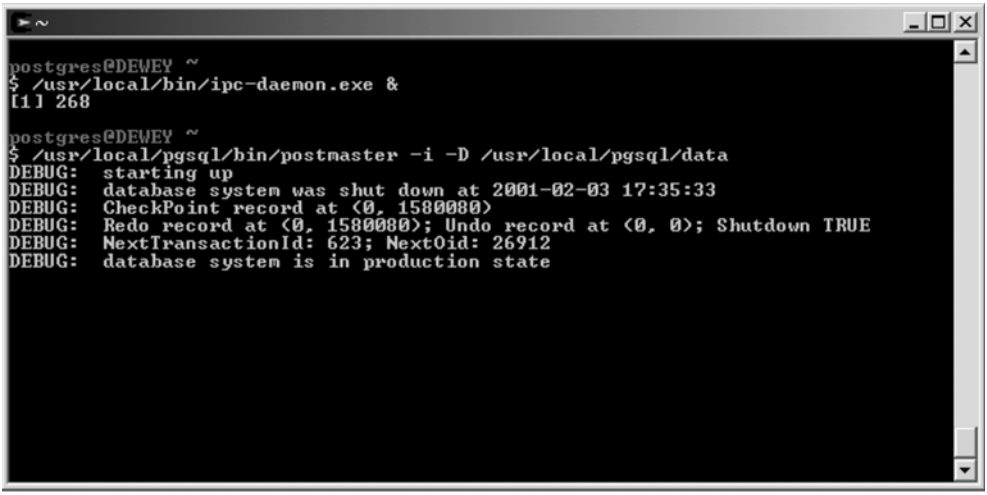


Рис. 3.13. Запуск PostgreSQL в Windows

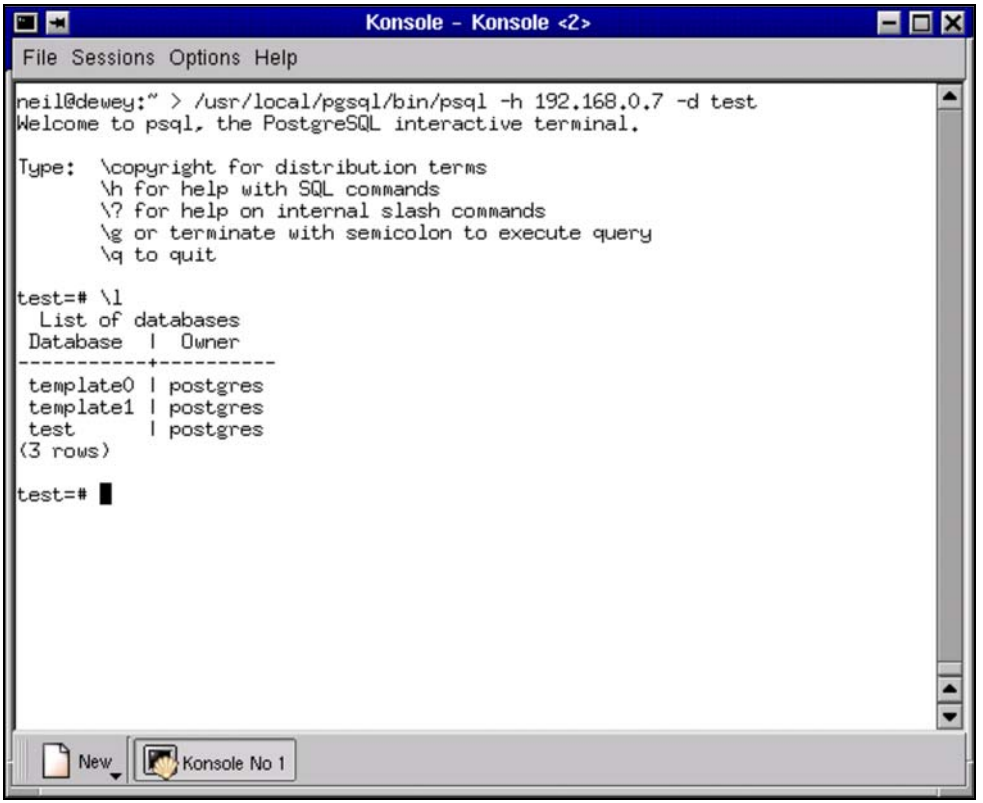


Рис. 3.14. Удаленный доступ к PostgreSQL с Linux-машины

Автоматический запуск PostgreSQL

Вот вы и научились запускать сервер PostgreSQL вручную. Теперь, если имеется компьютер под управлением Windows NT или 2000, выделенный под PostgreSQL, и вы готовы регистрироваться как пользователь `postgres` и запускать два процесса (`ipc-daemon` и `postmaster`) при каждом перезапуске системы, можно все оставить как есть.

Но если вы попробуете выйти из системы и войти как обычный пользователь, то обнаружите, что запущенные вами процессы остановлены.

Однако это можно исправить, заставив Windows автоматически запускать наши процессы. Для этого запустим их как службы Windows. В идеале программы должны быть специальным образом сконструированы, чтобы выполняться как службы, но даже если это не так, можно заставить Windows считать их таковыми.

Возможно, в будущих версиях PostgreSQL появится двоичный модуль, который будет устанавливаться как служба. Пожалуй, стоит следить за последними разработками, появляющимися на сайте.

А пока нам придется воспользоваться двумя небольшими программами: `INSTSRV` и `SRVANY`. Они входят в Windows NT4 Resource Kit и позволяют любую программу запустить в качестве службы. Они работают также и в Windows 2000. Исправленная версия `SRVANY` находится по адресу *ftp://ftp.microsoft.com/bussys/winnt/winnt-public/reskit/nt40/*.

Приведенные ниже инструкции требуют работы с редактором реестра. Редактирование реестра Windows – дело рискованное, т. к. цена ошибки может быть высока. Если вы не уверены в своих действиях, создайте резервную копию своих данных перед внесением изменений.

Скопируйте `instsrv.exe` и `srvany.exe` в любой локальный каталог. Программа `srvany.exe` всегда должна быть запущена во время старта служб PostgreSQL. Далее будем предполагать, что эти программы установлены в каталоге `C:\NTRESKIT`.

Создадим две новые службы в Windows, каждая из которых будет запускать утилиту `srvany.exe`, которая, в свою очередь, и запустит программу, выполняющуюся в качестве службы.

Начнем с более простой программы – демона IPC.

> ***Во время написания этой книги уже велись работы по модификации демона IPC, чтобы он мог запускаться как служба без помощи программ*** `INSTSRV` ***и*** `SRVANY`***. Если вы работаете с версией*** `cygipc` ***более поздней чем 1.09, обратитесь к файлу README за разъяснениями.***

Во-первых, необходимо создать новую службу, запустив утилиту `instsrv.exe`:

```
C:\NTRESKIT> instsrv "IPC Daemon"
c:\ntreskit\srvany.exe
```

```
The service was successfully added!

Make sure that you go into the Control Panel and use
the Services applet to change the Account Name and
Password that this newly installed service will use
for its Security Context.

C:\NTRESKIT>
```

Прежде чем редактировать реестр, добавляя туда параметры новой службы, необходимо настроить ее так, чтобы она запускалась под пользователем `postgres`. Для этого запустите оснастку Службы (Services) из состава средств Консоли управления (MMC).[1] В параметрах регистрации службы укажите имя пользователя `postgres` и пароль, выбранный ранее при создании пользователя (рис. 3.15). Поскольку пользователь зарегистрирован на локальной машине, а не в домене, то имя отображается в виде `.\postgres`, как это показано на рис. 3.15:

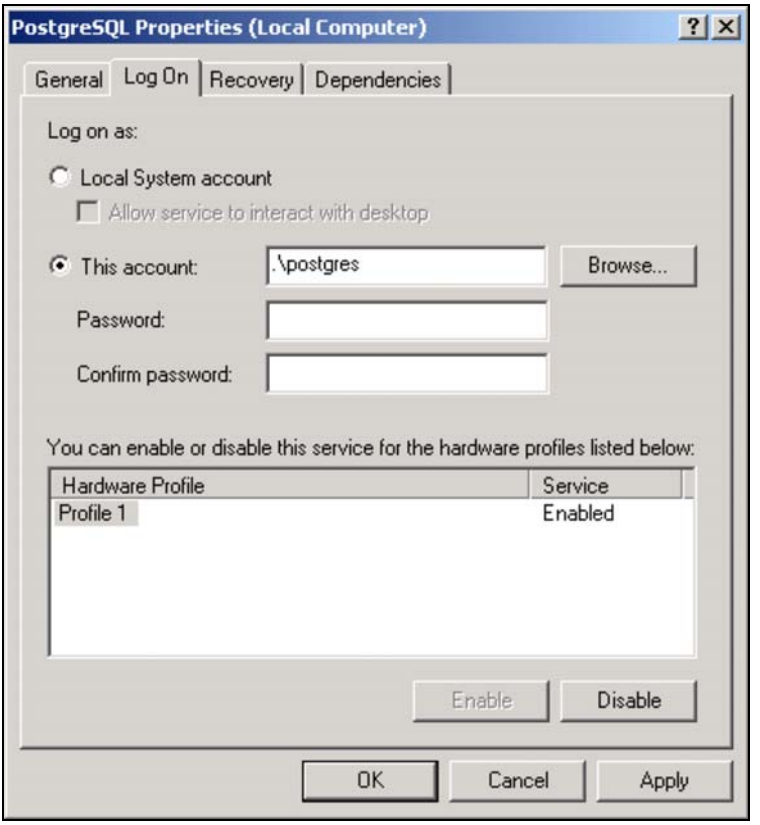


Рис. 3.15. Свойства PostgreSQL

[1] По цепочке Панель управления ▸ Администрирование ▸ Службы и приложения ▸ Службы. – *Примеч. ред.*

Теперь запустим редактор реестра `regedit.exe`, чтобы отредактировать запись о только что созданном демоне IPC. Необходимо указать параметры для утилиты `srvany.exe`, чтобы она запускала программу `ipc-daemon.exe` от нашего имени.

Запустите `regedit.exe` и перейдите в раздел

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services
```

Вы увидите новый элемент с именем `IPC Daemon`. Внутри него создайте ключ с именем `Parameters`, а в нем – два строковых параметра, `Application` и `AppDirectory`. Укажите в них полные пути программы `ipc-daemon.exe` и каталога `/bin`, используемого Cygwin.

Например, это могут быть `F:\cygwin\usr\local\bin\ipc-daemon.exe` и `F:\cygwin\bin` соответственно.

Результат должен выглядеть приблизительно так (рис. 3.16):

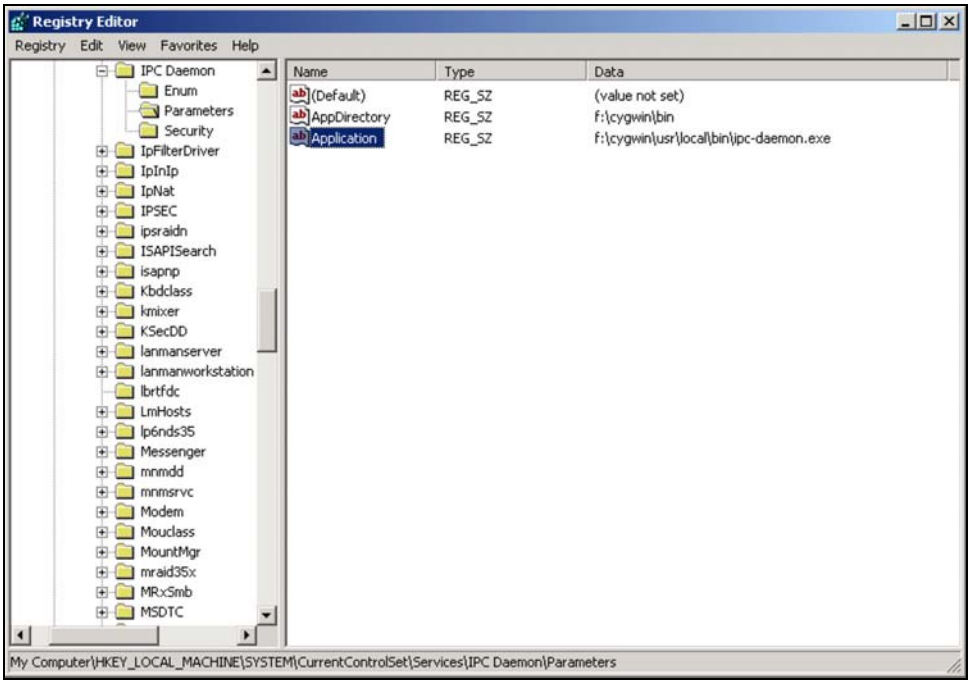


Рис. 3.16. Редактор реестра

Теперь можно запустить оснастку Службы (Services) из группы инструментов Администрирование (Administrative Tools) Панели управления (Control Panel) Windows 2000.

Программа `ipc-daemon.exe` должна стартовать, после чего она будет видна в окне Диспетчер задач Windows (Windows Task Manager) (рис. 3.17):

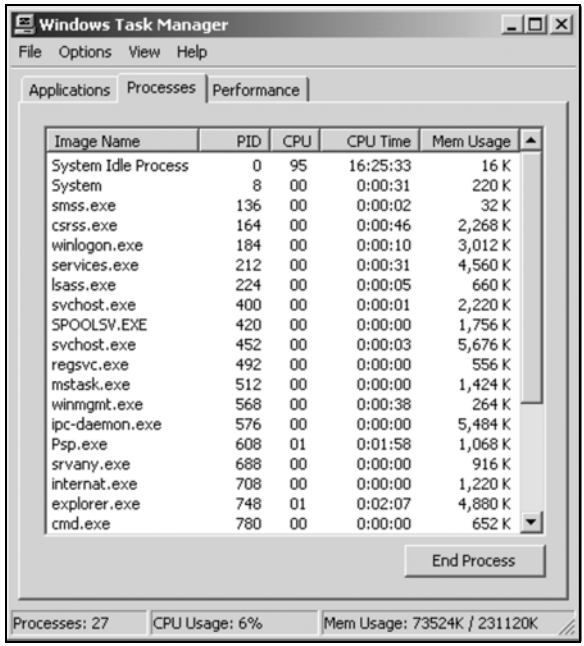


Рис. 3.17. Диспетчер задач Windows

Теперь перейдем к более сложной процедуре запуска процесса `postmaster`. Начнем также с создания службы `SRVANY`.

```
C:\NTRESKIT> instsrv "PostgreSQL" c:\ntreskit\srvany.exe
```

Прежде чем добавлять в реестр параметры для запуска этой службы, необходимо, как и для демона IPC, настроить ее запуск под пользователем `postgres`. Для этого запустите оснастку Службы и, как и раньше, укажите имя пользователя `.\postgres` и соответствующий пароль.

Для того чтобы запустить процесс `postmaster` в UNIX-подобной среде Cygwin, придется предварительно запустить пару командных процессоров. Используем `srvany.exe` для того, чтобы запустить `cmd.exe` – командную строку Windows. Ей передадим параметр, заставляющий ее запустить другую программу. Этой программой будет командный процессор Cygwin, который, в свою очередь, выполнит сценарий с командами оболочки. Этот-то сценарий и запустит процесс `postmaster`. Все не так уж сложно, если действовать последовательно.

Создайте сценарий для `bash` в каталоге Cygwin `/usr/local/bin` и назовите его `startpg`. Этот текстовый (в формате UNIX) файл должен содержать строки:

```
#!/bin/bash

# Start PostgreSQL

PGDATA=/usr/local/pgsql/data
PATH=/bin:/usr/bin:/usr/local/bin:/usr/local/pgsql/bin
export PGDATA PATH

postmaster -i >/usr/local/pgsql/postmaster.log 2>&1 </dev/null
```

Этот сценарий будет запускать PostgreSQL.

Теперь можно воспользоваться редактором `regedit.exe` для настройки параметров запуска. Как и раньше, перейдем в раздел `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services`. В нем должен появиться элемент `PostgreSQL`.

Создадим новый ключ с именем `Parameters`, а в нем – три строковых параметра со значениями, представленными в табл. 3.4:

Таблица 3.4. Компоненты ключа Parameters

Строка	Параметр
Application	`cmd.exe`
AppDirectory	`f:\cygwin\bin`
AppParameters	`/c "f:\cygwin\bin\bash.exe /usr/local/bin/startpg"`

Теперь можно запускать службу PostgreSQL. Используйте для этого оснастку Службы и диспетчер задач, чтобы убедиться, что обе службы запущены. Вы увидите, что задача `postmaster` в действительности называется `postgres.exe`, т. к. именно эту программу запускает сценарий `postmaster`.

В качестве завершающего теста перезагрузите машину: вы должны увидеть в диспетчере задач, что обе службы запущены и база данных PostgreSQL доступна для пользователя, отличного от `postgres`.

Пожалуй, это все.

Резюме

В данной главе рассмотрено несколько вариантов установки PostgreSQL. Самый простой способ – установка из предварительно откомпилированных пакетов. А поскольку PostgreSQL – продукт с открытым исходным кодом, то при необходимости он может быть откомпилирован пользователем в любой UNIX-совместимой системе.

Приведена последовательность шагов по компиляции, установке и проверке работоспособности с использованием пакетов в Linux, исходных текстов в UNIX и даже в Windows NT/2000 с Cygwin.

4

Доступ к данным

До сих пор наши встречи с SQL в этой книге носили непринужденный характер. Было рассмотрено несколько операторов для извлечения данных различными способами, а в предыдущей главе мы познакомились с операторами создания и заполнения таблиц. К этому моменту у вас уже должен работать сервер PostgreSQL. В следующей главе будет дано описание нескольких клиентских программ с графическим интерфейсом, а пока для доступа к базе данных будем использовать простую консольную программу `psql`.

В этой главе знакомство с SQL будет более формальным и начнется оно с оператора `SELECT`. На самом деле вся эта глава посвящена изучению оператора `SELECT`. У вас может сложиться впечатление, что целая глава об одном операторе – это перебор, но дело в том, что `SELECT` без преувеличения можно назвать основным оператором языка SQL. Если вы поняли, как работает этот оператор, можете считать, что большая часть пути по изучению SQL уже пройдена.

Изменять и удалять данные, а также применять оператор `SELECT` в выражениях для манипулирования данными мы научимся в последующих главах. Для наглядности будем писать ключевые слова SQL в верхнем регистре. Язык SQL не чувствителен к регистру, хотя в некоторых реализациях это не относится к названиям таблиц. Разумеется, в информации, хранящейся в базах данных, регистры различаются и символьная строка `'Newtown'` отличается от строки `'newtown'`.

Будем использовать программу `psql`, работающую с командной строкой, но все примеры этой главы можно выполнить в любой графической программе, позволяющей обращаться к PostgreSQL на языке SQL.

В этой главе мы рассмотрим:

- Использование psql
- Несколько простых выражений с оператором SELECT
- Замену названий столбцов
- Управление порядком строк
- Исключение повторяющихся строк
- Выполнение вычислений
- Более сложные условия
- Замещение названий таблиц
- Сравнение с образцом
- Сравнение данных разных типов
- Связывание данных в таблицах
- Объединение трех таблиц

В данной главе будем работать с учебной базой данных, спроектированной в главе 2 и созданной и заполненной данными в главе 3.

Использование psql

У тех, кто следовал инструкциям в главе 3, должна быть база данных с именем bpsimple, доступная обычному пользователю.

> ***Никогда не работайте с сервером PostgreSQL как пользователь*** postgres***, кроме случаев, требующих администрирования базы данных. Точно так же и в Linux, избегайте регистрироваться пользователем root, если только вам не требуются его привилегии для выполнения команды.***

Для того чтобы получить доступ к bpsimple с помощью программы psql, введите:

```
$ psql -d bpsimple
```

Вы должны увидеть:

```
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

bpsimple=#
```

Теперь можно вводить команды. Если psql выдаст ошибку на pg_shadow, то это означает, что вы не создали пользователя базы данных со своим именем. С помощью команды su станьте пользователем postgres,

выполните команду `createuser <имя пользователя>` и ответьте `'y'`. Закройте эту сессию, зарегистрируйтесь заново под своим именем и выполните команду `createdb`, чтобы создать базу данных `bpsimple`. Теперь, запустив `psql`, вы должны получить приглашение `bpsimple=#`. После этого придется выполнить действия, описанные в конце предыдущей главы, чтобы создать таблицы и данные, которые будут использованы в этой главе.

Чтобы убедиться в том, что таблицы созданы, введите `\dt` и нажмите клавишу <Enter>:

```
bpsimple=# \dt
                List of relations
            Name            |   Type   |  Owner
----------------------------+----------+----------
barcode                     | table    | rick
customer                    | table    | rick
item                        | table    | rick
orderinfo                   | table    | rick
orderline                   | table    | rick
stock                       | table    | rick

bpsimple=#
```

Имя `rick` в столбце `owner` будет заменено на ваше регистрационное имя.

В этой главе нам понадобятся лишь несколько основных команд `psql` (полный список приведен в главе 5), перечисленных в табл. 4.1, – каждая команда заканчивается нажатием клавиши <Enter>:

Таблица 4.1. Команды psql

Команда	Описание
`\?`	Вызвать подсказку
`\do`	Вывести список операторов
`\dt`	Вывести список таблиц
`\dT`	Вывести список типов
`\h <команда>`	Получить справку по SQL-команде
`\i <имя файла>`	Выполнить команды из файла
`\r`	Сбросить буфер (отменить ввод)
`\g`	Выйти из `psql`

В большинстве систем Linux для просмотра списка команд и их редактирования применяются клавиши со стрелками. Эта возможность в `psql` обеспечивается GNU-утилитой `readline`, которая в большинстве случаев установлена в системе.

Теперь можно использовать SQL для доступа к PostgreSQL. В следующей главе рассмотрим несколько программ с графическим интерфей-

сом, применяемых для работы с PostgreSQL, а пока ограничимся утилитой psql. Те, кто предпочитает графические программы, могут сначала заглянуть в главу 5, а потом вернуться сюда и использовать их вместо psql.

Простые выражения с оператором SELECT

Как и во всех реляционных СУБД, для извлечения данных в PostgreSQL предназначен SELECT. Не исключено, что это самый сложный оператор языка SQL, но именно он является ключом к эффективной работе с реляционными базами данных.

Начнем изучение оператора SELECT с простого запроса: показать все данные, хранящиеся в некоторой таблице. Для чего используем простейшую форму этого оператора, указав инструкцию FROM и имя таблицы:

```
SELECT <разделенный запятыми список столбцов> FROM <таблица>
```

Если вы забыли точные названия столбцов или хотите увидеть все столбцы, можете указать '*' вместо списка столбцов.

Попробуйте сами. Выбор всех столбцов из таблицы

Для начала выберем все столбцы из таблицы item:

```
SELECT * FROM item;
```

Помните, что ';' помогает программе psql определить, что ввод оператора закончен. Строго говоря, точка с запятой не является частью языка SQL. Можно заканчивать операторы и символами \g, которые имеют в psql точно такое же значение. В других программах для работы с SQL вам, возможно, не понадобится ни один из этих разделителей.

После нажатия <Enter> будет получен ответ:

```
bpsimple=# SELECT * FROM item;
 item_id |  description  | cost_price | sell_price
---------+---------------+------------+------------
       1 | Wood Puzzle   |      15.23 |      21.95
       2 | Rubic Cube    |       7.45 |      11.49
       3 | Linux CD      |       1.99 |       2.49
       4 | Tissues       |       2.11 |       3.99
       5 | Picture Frame |       7.54 |       9.95
       6 | Fan Small     |       9.23 |      15.75
       7 | Fan Large     |      13.36 |      19.95
       8 | Toothbrush    |       0.75 |       1.45
       9 | Roman Coin    |       2.34 |       2.45
      10 | Carrier Bag   |       0.01 |       0.00
      11 | Speakers      |      19.73 |      25.32
(11 rows)
bpsimple=#
```

Как это работает

Мы просто попросили PostgreSQL показать все данные из всех столбцов таблицы `item`, указав шаблон `'*'` вместо списка столбцов, а программа PostgreSQL эту просьбу выполнила, аккуратно напечатав названия столбцов и разделив их символами `'|'`. Она даже сообщила нам, сколько строк получено.

Это работает, но что если мы не хотим видеть все столбцы? Вообще говоря, у PostgreSQL, как и у любой другой СУБД, следует запрашивать только те данные, которые действительно нужны. Каждое поле каждой строки в запросе – это небольшая дополнительная нагрузка на сервер. Не забывайте стремиться к краткости и эффективности.

Когда вы начнете использовать SQL из других языков программирования (см. главу 14), то обнаружите, что явное указание столбцов позволяет работать с изменяющейся схемой базы данных. Например, если вы задаете `'*'`, а в таблицу добавлены новые поля, то вы можете обнаружить, что обрабатываете вовсе не те данные, которые собирались. Если необходимый столбец удален, то оператор SQL завершится с ошибкой, т. к. не сможет получить запрашиваемые данные – но такую ошибку значительно проще найти и исправить, чем ошибку в обработке данных, вызванную смещением столбцов.

И, разумеется, если мы обращаемся к столбцам по именам, у нас всегда есть возможность найти их в тексте программы и предотвратить ошибку.

Попробуем теперь ограничить количество запрашиваемых столбцов. Как было показано выше, для этого надо перечислить имена столбцов, разделяя их запятыми. Если требуется получить столбцы в порядке, отличном от того, в котором они были указаны при создании таблицы, достаточно перечислить их в нужной последовательности.

Попробуйте сами. Выбор столбцов по имени в указанном порядке

Для выборки названия города и фамилии клиента укажем имена соответствующих столбцов и, разумеется, имя таблицы. Вот оператор и ответ PostgreSQL:

```
bpsimple=# SELECT town, lname FROM customer;
   town    |  lname
-----------+---------
 Hightown  | Stones
 Lowtown   | Stones
 Nicetown  | Matthew
 Yuleville | Matthew
 Oahenham  | Cozens
 Nicetown  | Matthew
 Bingham   | Stones
 Bingham   | Stones
```

```
 Histon    | Hickman
 Tibsville | Howard
 Bingham   | Jones
 Winersby  | Neill
 Oxbridge  | Hendy
 Welltown  | O'Neill
 Milltown  | Hudson
(15 rows)

bpsimple=#
```

Как это работает

PostrgreSQL возвращает данные из всех строк указанной таблицы, но только из указанных столбцов. Столбцы выводятся в том порядке, в каком они были указаны в операторе `SELECT`.

Замена названий столбцов

Читатели, вероятно, уже заметили, что название столбца в базе данных выступает в качестве заголовка при выводе. Иногда это не очень удобно, особенно когда (как будет показано в главе 7) выводимый столбец не является столбцом таблицы и, следовательно, не имеет имени. Назначить столбцу имя для вывода очень просто, достаточно добавить конструкцию `AS <отображаемое имя>` после каждого столбца в операторе `SELECT`. Можно назначить имена всем выбираемым столбцам, или же только некоторым из них. Там, где имя не указано, подставляется имя столбца таблицы.

Так, для того чтобы в предыдущем примере заменить `lname` на `Last Name`, введите

```
SELECT town, lname AS "Last Name" FROM customer;
```

В следующем разделе мы увидим это в действии.

Изменение порядка строк

До сих пор мы получали данные из нужных столбцов, но они не всегда следовали в порядке, удобном для просмотра. Выводимые данные могут располагаться в том порядке, в каком они были вставлены в таблицу, но в действительности, если в запросе нет явных указаний, реляционные базы данных возвращают строки в произвольном порядке.

В главе 2 уже упоминалось о том, что, в отличие от электронных таблиц, порядок строк в базах данных не определен. Сервер стремится расположить данные наиболее эффективным образом, не заботясь об удобстве их просмотра. Строки выводятся неотсортированными, и при следующем запросе тех же данных порядок их вывода может измениться.

Обычно данные выводятся в порядке, соответствующем внутреннему расположению их в базе данных. Он может совпадать с порядком

вставки строк в таблицу, но если строки удаляются, а затем вставляются новые, то расположение строк выглядит все более и более случайным. Ни одна SQL-совместимая база данных, включая PostgreSQL, не обязана возвращать данные в каком-либо определенном порядке, если явно не задано условие сортировки.

Управление порядком строк, возвращаемых оператором SELECT, осуществляется добавлением к нему инструкции ORDER BY, в которой определяется требуемый порядок сортировки. Синтаксис выражения:

```
SELECT <разделенный запятыми список столбцов> FROM <таблица> ORDER BY
<столбец> [ASC | DESC]
```

После имени столбца можно указать параметры ASC (от ascending – возрастающий) или DESC (от descending – убывающий). По умолчанию выполняется сортировка по возрастанию. Данные возвращаются отсортированными по значению указанного столбца в указанном порядке.

Попробуйте сами. Сортировка данных

В этом примере отсортируем данные по названию города и заменим название столбца lname, как было показано в предыдущем разделе, на более удобочитаемое.

Обратите внимание, что, поскольку нам требуется сортировка по возрастанию, можно пропустить параметр ASC, т. к. он действует по умолчанию. Вот команда и ответ PostgreSQL:

```
bpsimple=# SELECT town, lname AS "Last Name" FROM customer ORDER BY town;
   town    | Last Name
-----------+-----------
 Bingham   | Stones
 Bingham   | Stones
 Bingham   | Jones
 Hightown  | Stones
 Histon    | Hickman
 Lowtown   | Stones
 Milltown  | Hudson
 Nicetown  | Matthew
 Nicetown  | Matthew
 Oahenham  | Cozens
 Oxbridge  | Hendy
 Tibsville | Howard
 Welltown  | O'Neill
 Winersby  | Neill
 Yuleville | Matthew
(15 rows)

bpsimple=#
```

Как видите, данные теперь отсортированы по названию города в порядке возрастания.

Как это работает

На этот раз внесено два изменения в первоначальный оператор. Добавлена конструкция AS для замены названия второго столбца на Last Name, более удобного для восприятия, и инструкция ORDER BY, чтобы определить порядок, в котором PosgteSQL выводит данные.

Иногда приходится идти дальше и сортировать данные более чем по одному столбцу. Так, в приведенном примере данные отсортированы по полю town, но в столбце Last Name никакого порядка не наблюдается. В частности, видно, что среди клиентов из города Bingham клиент Jones следует за клиентом Stones.

Указав более одного поля в ORDER BY, можно более аккуратно рассортировать данные. При желании можно даже указать сортировку по возрастанию для одного столбца и по убыванию – для другого.

Попробуйте сами. Сортировка данных с параметрами ASC и DESC

Выполним оператор SELECT еще раз, но теперь отсортируем названия городов по убыванию, а имена – по возрастанию в пределах одного города. На этот раз не будем заменять название столбца.

Вот оператор и ответ PostgreSQL на него:

```
bpsimple=# SELECT town, lname FROM customer ORDER BY town DESC, lname ASC;
town      | lname
----------+-----------
Yuleville | Matthew
Winersby  | Neill
Welltown  | O'Neill
Tibsville | Howard
Oxbridge  | Hendy
Oahenham  | Cozens
Nicetown  | Matthew
Nicetown  | Matthew
Milltown  | Hudson
Lowtown   | Stones
Histon    | Hickman
Hightown  | Stones
Bingham   | Jones
Bingham   | Stones
Bingham   | Stones
(15 rows)

bpsimple=#
```

Как это работает

Как видите, данные сначала были отсортированы в порядке убывания по значению поля town, указанного первым в инструкции ORDER BY. Затем там, где встречаются несколько имен в одном городе, записи были

отсортированы по возрастанию. Теперь, хотя город Bingham находится в конце списка, фамилии наших клиентов в этом городе упорядочены по алфавиту.

Вполне естественно, что обычно список полей, по которым можно выполнять сортировку, ограничен полями, данные из которых запрошены. Что касается PostgreSQL, то, по крайней мере в текущей версии, это ограничение не действует, и в инструкции `ORDER BY` можно указывать столбцы, не участвующие в выборке данных. Это отступление от стандарта SQL, и мы настоятельно рекомендуем не использовать такую возможность.

Исключение повторяющихся строк

Вы могли заметить, что некоторые строки в вышеприведенном примере повторяются дважды, в частности эти:

```
Nicetown  | Matthew
Bingham   | Stones
```

Почему это происходит? Если обратиться к исходным данным в главах 2 и 3, то видно, что действительно в городе Nicetown есть два клиента по фамилии Matthew, два клиента из города Bingham имеют фамилию Stones. Приведем для справки соответствующие строки с указанием их имен:

```
Nicetown  | Matthew | Alex
Nicetown  | Matthew | Neil
Bingham   | Stones  | Richard
Bingham   | Stones  | Ann
```

Когда PostgreSQL выбрала две строки для Nicetown и Matthew и две – для Bingham и Stones, то поступила вполне корректно. В каждом из этих городов есть по два клиента с одинаковыми фамилиями.

Записи выглядят одинаково, потому что не выведены поля, в которых они отличаются. По умолчанию будут выведены все строки, но это не всегда то, что требуется.

Предположим, что нам нужен список городов, в которых есть наши клиенты, – например, для того, чтобы решить, где стоит открывать отделения компании. Воспользовавшись уже полученными знаниями, попробуем такой путь:

```
bpsimple=# SELECT town FROM customer ORDER BY town;
   town
-----------
 Bingham
 Bingham
 Bingham
 Hightown
```

```
 Histon
 Lowtown
 Milltown
 Nicetown
 Nicetown
 Oahenham
 Oxbridge
 Tibsville
 Welltown
 Winersby
 Yuleville
(15 rows)

bpsimple=#
```

Получены правильные данные, но, вероятно, не совсем те, которые хотелось получить.

Для каждой строки в таблице `customer` PostgreSQL вывела название города. Это правильный список, но нужен немного другой. А требуется нам список, в который каждый город входит лишь один раз, другими словами – список городов с различающимися названиями.

Язык SQL позволяет подавить вывод повторяющихся строк с помощью инструкции `DISTINCT` в операторе `SELECT`. Синтаксис оператора:

```
SELECT DISTINCT <разделенный запятыми список столбцов> FROM <таблица>
```

Как и большинство других инструкций в операторе `SELECT`, его можно комбинировать с другими, например с инструкциями переименования столбцов или сортировки.

Попробуйте сами. Использование DUISTINCT

Выведем список всех городов, встречающихся в таблице `customer`, без повторений. Попробуем такой способ:

```
bpsimple=# SELECT DISTINCT town FROM customer;
   town
-----------
 Bingham
 Hightown
 Histon
 Lowtown
 Milltown
 Nicetown
 Oahenham
 Oxbridge
 Tibsville
 Welltown
 Winersby
```

```
 Yuleville
(12 rows)

bpsimple=#
```

Как это работает

Инструкция DISTINCT заставляет PostgreSQL удалять все повторяющиеся строки. Обратите внимание, что в этом случае список упорядочен. Это происходит благодаря алгоритму, который PostgreSQL использует для отбора данных. Вообще, данные не обязаны быть отсортированы таким образом. Для того чтобы получить упорядоченные каким-либо способом данные, следует явно определить способ сортировки при помощи ORDER BY.

Имейте в виду, что инструкция DISTINCT не связана с каким-то определенным столбцом. Можно исключать только строки, совпадающие по всем выбранным полям, но исключить их по какому-то одному выбранному полю не удастся. Рассмотрим такой оператор:

```
SELECT DISTINCT town, fname FROM customer;
```

Здесь опять получено 15 строк, т. к. существуют 15 различных комбинаций названия города и имени.

Здесь хотим вас предостеречь. Хотя может показаться, что есть резон всегда исключать повторения в операторе SELECT, на самом деле это плохая идея – по двум причинам. Прежде всего, инструкция DISTINCT требует от PostgreSQL значительно большей работы по отбору данных и проверке на совпадение. Не используйте DISTINCT, если только вы не уверены, что повторяющиеся строки должны быть удалены. Вторая причина несколько прагматичнее. Иногда DISTINCT может маскировать ошибки в данных или в SQL-запросах, которые без труда могли бы быть обнаружены по повторяющимся строкам. В общем, применяйте DISTINCT только тогда, когда это действительно необходимо.

Выполнение вычислений

Над выбранными данными, прежде чем выводить их, можно проделать несложные вычисления.

Предположим, что требуется вывести себестоимость товаров из таблицы item. Можно использовать такой оператор SELECT:

```
bpsimple=# SELECT description, cost_price FROM item;
  description  |  cost_price
---------------+--------------
 Wood Puzzle   |       15.23
 Rubic Cube    |        7.45
 Linux CD      |        1.99
 Tissues       |        2.11
```

```
 Picture Frame |       7.54
 Fan Small     |       9.23
 Fan Large     |      13.36
 Toothbrush    |       0.75
 Roman Coin    |       2.34
 Carrier Bag   |       0.01
 Speakers      |      19.73
(11 rows)

bpsimple=#
```

Теперь предположим, что нас интересует себестоимость, выраженная в центах. Тогда, проделав в SQL-операторе простые вычисления, получим:

```
bpsimple=# SELECT description, cost_price *100 FROM item;
  description  |  cost_price
---------------+---------------
 Wood Puzzle   |  1523.00
 Rubic Cube    |   745.00
 Linux CD      |   199.00
 Tissues       |   211.00
 Picture Frame |   754.00
 Fan Small     |   923.00
 Fan Large     |  1336.00
 Toothbrush    |    75.00
 Roman Coin    |   234.00
 Carrier Bag   |     1.00
 Speakers      |  1973.00
(11 rows)

bpsimple=#
```

Десятичная точка выглядит здесь немного странно, поэтому избавимся от нее при помощи трюка, к которому еще вернемся позднее.

Используем функцию `CAST` для изменения типа столбца:

```
bpsimple=# SELECT description, CAST(cost_price *100 AS INT) FROM item;
description   |  cost_price
--------------+---------------
Wood Puzzle   |       1523
Rubic Cube    |        745
Linux CD      |        199
Tissues       |        211
Picture Frame |        754
Fan Small     |        923
Fan Large     |       1336
Toothbrush    |         75
Roman Coin    |        234
Carrier Bag   |          1
Speakers      |       1973
(11 rows)

bpsimple=#
```

Вычисления над данными столбцов требуются достаточно редко. Если они и применяются, то, как правило, при изменении данных в базе, как будет показано в главе 7. Однако не лишним будет знать, что такая возможность существует.

Выбор строк

До сих пор в этой главе работа велась либо со всеми имеющимися строками данных, либо со всеми различными. Теперь посмотрим, как, подобно столбцам, можно выбирать нужные нам строки. Вас, вероятно, не удивит, что для этого применяется еще одна инструкция в операторе SELECT.

Это новая инструкция WHERE, синтаксис которой выглядит следующим образом:

```
SELECT <разделенный запятыми список столбцов> FROM <таблица> WHERE <условие>
```

Существует множество условий, которые могут быть объединены операторами AND, OR или NOT.

Стандартный список операторов сравнения приведен в табл. 4.2:

Таблица 4.2. Операторы сравнения

Оператор	Описание
<	Меньше
<=	Меньше или равно
=	Равно
>=	Больше или равно
>	Больше
<>	Не равно

Эти операторы могут применяться с большинством типов данных, включая числовые и строковые, хотя позже вы узнаете, что для работы с датами вводится ряд специальных условий.

Начнем с простых условий и выберем записи о людях, проживающих в городе Bingham. Для этого используем такое выражение:

```
bpsimple=# SELECT town, lname, fname FROM customer WHERE town = 'Bingham';
 town    | lname  |  fname
---------+--------+--------
Bingham | Stones | Richard
Bingham | Stones | Ann
Bingham | Jones  | Dave
(3 rows)

bpsimple=#
```

Это оказалось совсем просто, правда? Обратите внимание на одинарные кавычки, в которые заключено слово «Bingham» – они необходимы, чтобы данные интерпретировались как строка символов. Заметим также, что, поскольку строка «Bingham» сравнивается с данными, то регистр имеет значение. Если написать ... `town = 'bingham'`, то ни одна строка не будет выбрана, т. к. операция сравнения строк чувствительна к регистру.

Можно задать несколько условий, объединяя их операторами `AND`, `OR`, `NOT` и используя скобки. В условиях можно указывать и те поля, которые отсутствуют в списке выбираемых – в отличие от инструкции `ORDER BY`, где это недопустимо.

Попробуйте сами. Использование операторов

Применим более сложную комбинацию условий. Предположим, нам требуется выбрать имена клиентов, живущих в Bingham или в Nicetown, к которым не следует обращаться «мистер».

Вот соответствующий оператор и возвращаемые им данные:

```
bpsimple=# SELECT title, fname, lname, town FROM customer WHERE title <> 'Mr'
bpsimple-# AND (town = 'Bingham' OR town = 'Nicetown');
title  | fname |  lname  |   town
-------+-------+---------+----------
 Miss  | Alex  | Matthew | Nicetown
 Mrs   | Ann   | Stones  | Bingham
(2 rows)

bpsimple=#
```

Как это работает

На первый взгляд оператор может показаться сложным, но на самом деле он довольно прост. Первая его часть – это обычный `SELECT` со списком интересующих нас полей. После ключевого слова `WHERE` сначала проверяем, что к клиенту не надо обращаться «мистер» (Mr), затем проверяем, истинно ли второе условие, и используем `AND`. Второе условие определяет, что это за город – Bingham или Nicetown. Обратите внимание, что для явного задания порядка выполнения операций сравнения применяются скобки.

Следует иметь в виду, что PostgreSQL, как и все прочие реляционные базы данных, не обязан выполнять инструкции в том порядке, в каком они встречаются в операторе SQL. Гарантируется только, что результат его выполнения будет правильным ответом на заданный «SQL-вопрос». Обычно реляционные СУБД используют сложный оптимизатор, который, анализируя запросы, вычисляет наилучший способ их выполнения. Оптимизаторы не совершенны, и вы можете однажды обнаружить, что оператор выполняется быстрее, если его переписать другим

способом. Для несложных операторов, аналогичных рассмотренному выше, можно считать, что оптимизатор выполняет полезную работу.

Если вы хотите знать, как PostgreSQL станет обрабатывать оператор SQL, начните оператор SQL со слова «explain» (объяснить). PostgreSQL не станет выполнять такой оператор, а выведет разъяснения о том, как он будет это делать.

Более сложные условия

Одной из распространенных задач при работе со строками является сравнение с образцом. Например, вы ищете человека по имени Роберт, но в базе данных его имя может быть записано как Роб или даже Боб. В SQL есть несколько специальных операций, облегчающих работу со строками, подстроками и списками строк.

Первое из этих условий – `IN`, позволяющее выполнить сравнение со списком значений без применения громоздкой конструкции с условием `OR`. Рассмотрим пример:

```
SELECT title, fname, lname, town FROM customer WHERE title <> 'Mr' AND (town =
'Bingham' OR town = 'Nicetown');
```

Это можно переписать как:

```
SELECT title, fname, lname, town FROM customer WHERE title <> 'Mr' AND town IN
('Bingham', 'Nicetown');
```

Результат будет таким же, хотя, возможно, порядок строк изменится, поскольку мы обошлись без инструкции `ORDER BY`. В данном случае применение `IN` не дает заметных преимуществ, кроме упрощения записи. Позже, познакомившись с вложенными запросами, мы вернемся к `IN`, и тогда его преимущество станет более заметным.

Следующее условие – `BETWEEN`. Оно позволяет проверить принадлежность значения диапазону, заданному своими крайними точками. Предположим, что нужны те строки таблицы, в которых значение `customer_id` находится между 5 и 9. Вместо последовательности условий `OR` или условия `IN` с длинным списком значений можно использовать более простую запись:

```
bpsimple=# SELECT customer_id, town, lname FROM customer WHERE customer_id
bpsimple-# BETWEEN 5 AND 9;
 customer_id |   town    |  lname
-------------+-----------+---------
           5 | Oahenham | Cozens
           6 | Nicetown | Matthew
           7 | Bingham  | Stones
           8 | Bingham  | Stones
           9 | Histon   | Hickman
(5 rows)
bpsimple=#
```

Условие BETWEEN можно использовать и для строк, но следует быть внимательным при задании границ, чтобы не получить неожиданный ответ, и, разумеется, надо помнить о регистре – как уже говорилось, операция сравнения строк чувствительна к нему.

Попробуйте сами. Сложные условия

Применим инструкцию BETWEEN для сравнения строк. Предположим, нам нужен список городов, названия которых начинаются с буквы, расположенной между B и N. Нам известно, что все названия городов в нашей базе данных начинаются с заглавной буквы, поэтому можно написать:

```
bpsimple=# SELECT DISTINCT town FROM customer
bpsimple-# WHERE town BETWEEN 'B' AND 'N';
   town
-----------
 Bingham
 Hightown
 Histon
 Lowtown
 Milltown
(5 rows)

bpsimple=#
```

При внимательном рассмотрении результат оказывается не таким, которого мы ожидали. Где город Newtown? Его название начинается на N, но в списке он отсутствует.

Почему это не работает

Причина, по которой этот оператор не работает, заключается в том, что PostgreSQL, в соответствии со стандартом SQL92, дополняет переданную ему строку пробелами до тех пор, пока ее длина не сравняется с длиной сравниваемой строки. Поэтому, когда очередь доходит до «Newtown», PostgreSQL сравнивает «N» (с последующими шестью пробелами) и «Newtown», а поскольку пробелы в кодировке ASCII расположены раньше всех остальных символов, он решает, что строка «Newtown» следует после «N» и не должна быть включена в список.

Как заставить это работать

Заставить работать этот оператор достаточно просто. Следует либо предотвратить дополнение строки пробелами, самостоятельно добавив к ней несколько символов «z», либо в условии BETWEEN использовать следующую букву – «O». Правда, если встретится город с названием «O», то он ошибочно будет включен в выборку, поэтому такой способ следует применять с осторожностью. Буква «z» располагается в ASCII-

таблице после заглавной «Z», поэтому лучше использовать ее. Таким образом, получаем:

```
SELECT DISTINCT town FROM customer WHERE town BETWEEN 'B' AND 'Nz';
```

Заметьте, что мы не добавили букву «Z» после «B», т. к. строка «B», дополненная пробелами, подходит для поиска любых названий, начинающихся с буквы «B». Опять-таки, если встретится город с названием «Nzz», он не будет найден, потому что сравнение «Nz» и «Nzz» покажет, что «Nzz» следует за «Nz» – поскольку строка «Nz» будет дополнена пробелом, а он встречается раньше, чем символ «z» в третьей позиции строки, с которой выполняется сравнение.

При таком способе сравнения результат не всегда очевиден, и вероятность ошибки велика даже для опытного программиста на SQL. Поэтому рекомендуем не применять `BETWEEN` для сравнения строк.

Поиск по шаблону

Сами по себе рассмотренные операции сравнения строк вполне работоспособны, но от них не много проку в практических задачах сравнения с образцом. Для таких случаев в SQL предусмотрено условие `LIKE`.

К сожалению, `LIKE` руководствуется при сравнении строк таким набором правил, который не встречается больше ни в одном известном нам языке программирования. Но если запомнить эти правила, то пользоваться ими очень легко. При сравнении строк посредством `LIKE` знак процента (`%`) означает произвольную строку символов, а знак подчеркивания (`_`) соответствует одному любому символу.

Например, чтобы выбрать города, названия которых начинаются с буквы «B», напишем:

```
… WHERE town LIKE  'B%'
```

А для выбора имен, оканчивающихся на «e», напишем:

```
… WHERE fname LIKE '%e';
```

Для выбора имен, состоящих из четырех букв, используем четыре знака подчеркивания:

```
… WHERE fname LIKE '_ _ _ _';
```

При необходимости оба шаблона можно использовать в одной строке.

Попробуйте сами. Поиск по шаблону

Найдем всех клиентов, имена которых во второй позиции содержат букву «a».

Используем такой оператор:

```
bpsimple=# SELECT fname, lname FROM customer WHERE fname LIKE '_a%';
fname | lname
------+--------
Dave  | Jones
Laura | Hendy
David | Hudson
(3 rows)

bpsimple=#
```

Как это работает

Первая часть шаблона `_a` соответствует строке, в которой первый символ может быть любым, а второй должен быть буквой «а» нижнего регистра. Вторая часть, `%`, соответствует любым оставшимся символам. Если не использовать `%`, то шаблону будут соответствовать только строки из двух букв.

Ограничение количества выводимых строк

В приводимых до сих пор примерах количество строк, возвращаемых запросом, невелико, т. к. используется «экспериментальная» база данных, содержащая лишь несколько записей. В «настоящих» базах данных вполне могут храниться многие тысячи записей, соответствующих нашему критерию отбора, и, если заниматься отладкой SQL-операторов, то наблюдать, как тысячи строк пробегают по экрану, будет весьма утомительно. Нескольких пробных строк для проверки логики вполне достаточно.

СУБД PostgreSQL позволяет использовать в операторе `SELECT` дополнительную инструкцию `LIMIT`, которая не входит в стандарт языка SQL, но бывает очень полезна, когда требуется ограничить количество выводимых строк.

Инструкция `LIMIT` может применяться двумя немного различными способами. Если к оператору `SELECT` добавить `LIMIT` и число, то будут выведены строки с номерами от первого до указанного числа.

Другой вариант использования `LIMIT` предполагает указание двух чисел, разделенных запятой. Если написать `LIMIT M, N`, то первые `N` строк будут пропущены, а выведены будут следующие за ними `M` строк.

Проще показать это на примере, чем описывать на словах. Здесь выводятся только первые пять строк:

```
bpsimple=# SELECT customer_id, town FROM customer LIMIT 5;
customer_id  |   town
-------------+-----------
           1 | Hightown
           2 | Lowtown
```

```
           3 | Nicetown
           4 | Yuleville
           5 | Oahenham
(5 rows)

bpsimple=#
```

А здесь пропускаются первые две строки, затем пять строк выводятся:

```
bpsimple=# SELECT customer_id, town FROM customer LIMIT 5,2;
 customer_id |   town
-------------+-----------
           3 | Nicetown
           4 | Yuleville
           5 | Oahenham
           6 | Nicetown
           7 | Bingham
(5 rows)

bpsimple=#
```

Если инструкция `LIMIT` используется в операторе `SELECT` совместно с другими, то она должна быть последней.

Сравнение других типов данных

Есть два особых случая сравнения, требующих отдельного рассмотрения. В главе 2 был рассмотрен специальный тип `NULL`, соответствующий «неопределенному» или «нерелевантному» значению. Рассмотрим это подробнее, т. к. если в столбце встречается значение `NULL`, то при сравнении следует предпринять некоторые действия, чтобы результат соответствовал ожидаемому.

Придется также забежать немного вперед и выяснить, как выполняется сравнение данных, предназначенных в PostgreSQL для хранения даты и времени.

Сравнение с NULL

До сих пор нам не известно, как проверить, содержит ли столбец значение `NULL`. Можно проверить, равняется ли его значение числу или строке или нет, но этого недостаточно. Предположим, что в таблице `testtab` имеется столбец целых чисел `tryint`, о котором известно, что он содержит значения либо `0`, либо `1`, либо `NULL`. Проверим равенство нулю:

```
SELECT * FROM testtab WHERE tryint = 0;
```

Проверим равенство единице:

```
SELECT * FROM testtab WHERE tryint = 1;
```

Но для того чтобы сравнить с NULL, необходим еще один вид проверки. Стандарт SQL определяет синтаксис для проверки значения поля на равенство значению NULL, и PostgreSQL его поддерживает. Выполним проверку, применив конструкцию IS NULL:

```
SELECT * FROM testtab WHERE tryint IS NULL;
```

Заметьте, что здесь использовано ключевое слово IS, а не знак равенства. Можно также проверить, что значение отлично от NULL, добавив NOT для инвертирования условия:

```
SELECT * FROM testtab WHERE tryint IS NOT NULL;
```

Почему нам вдруг понадобилась такая дополнительная синтаксическая конструкция? Дело в том, что вместо привычной двузначной логики, где допустимы только значения ИСТИНА и ЛОЖЬ, мы столкнулись с трехзначной логикой, оперирующей значениями ИСТИНА, ЛОЖЬ и НЕ ОПРЕДЕЛЕНО.

К сожалению, это свойство «неопределенности» значения NULL имеет некоторые побочные эффекты при выполнении сравнения.

Пусть наш оператор выполняется на таблице, в которой столбец tryint содержит несколько значений NULL:

```
SELECT * FROM testtab WHERE tryint = 1;
```

Что означает выражение tryint = 1, когда в действительности значение поля равно NULL? Вопрос о равенстве неопределенного значения единице действительно интересен, т. к. на него невозможно дать ни отрицательный, ни утвердительный ответ. А раз ответ неизвестен, то строки, содержащие NULL, не соответствуют условию. Если изменить условие на противоположное и написать tryint = 0, строки опять не будут выбраны, поскольку и это условие не выполняется. Это может сбить с толку, ведь две проверки с противоположными условиями так и не позволили получить все строки таблицы.

Эти особенности значения NULL необходимо постоянно иметь в виду. Если в условиях фигурируют столбцы, которые могут иметь неопределенные значения, и получаются странные результаты, – проверьте, не является ли NULL источником проблем.

Сравнение дат и времени

Мы познакомились еще не со всеми типами данных PostgreSQL, а для работы с датой и временем понадобятся несколько специальных функций.

В PostgreSQL имеются два основных типа для даты и времени: TIMESTAMP, содержащий и дату и время, и DATE – для хранения дня, месяца и года. Несколько встроенных функций PostgreSQL предназначены для

манипулирования временем и датами, традиционно именно эти данные чаще всего вызывают затруднения. Не будем рассматривать здесь все имеющиеся функции (они описаны в документации PostgreSQL), остановимся только на самых необходимых.

Прежде чем начать, нам надо разобраться с одной из тех мелких «неувязочек», которые легко могут привести к неразберихе. Как записывать дату?

Что мы подразумеваем, когда пишем 1/2/1997? Для европейцев это первое февраля 1997 года, а для американцев – второе января. Дело в том, что в Европе принято записывать дату в формате DD/MM/YYYY, а в Америке – в формате MM/DD/YYYY. Нас всегда интересовало, почему так получилось, но эта тема не для данной книги.

Путаница с датами создает некоторые неудобства, когда мы просим PostgreSQL преобразовать подобную строку в значение даты. Поэтому PostgreSQL позволяет изменить правила преобразования дат в соответствии с местными стандартами, и прежде, чем оперировать датой и временем, будет полезно разобраться, каким образом можно управлять этим аспектом поведения системы.

Задание стиля даты и времени

К сожалению, метод, принятый в PostgreSQL для определения способа интерпретации времени и даты, выглядит несколько странным – по крайней мере на первый взгляд.

Есть две вещи, которыми можно управлять:

- Порядок следования дней и месяцев: американский или европейский стиль
- Формат отображения: числовой или текстовый

Неприятность заключается в том, что в PostgreSQL для управления обоими параметрами используется одна переменная, что, честно говоря, может немного запутать.

Хорошо, однако, что по умолчанию PostgreSQL руководствуется однозначно определенным в стандарте ISO-8601 форматом представления времени и даты: YYYY-MM-DD hh:mm:ss.sTZD. Например, полная запись даты и времени может выглядеть как 1997-02-01 05:23:43.23+5, что означает 1 февраля 1997 года, 5 часов 23 минуты и 43,23 секунды. Сочетание TZD в конце строки формата – это указатель часового пояса (Time Zone Designator), который мы пока не рассматриваем.

При вводе даты в формате NN/NN/NNNN PostgreSQL по умолчанию предполагает американский стиль, другими словами, 2/1/97 будет интерпретироваться как 1 февраля. Также можно использовать формат вида «February 1, 1997» либо формат ISO «1997-02-01». Если вам достаточно этих форматов, то и хорошо, – не придется разбираться с тем, как PostgreSQL вводит и отображает даты, и мы можем перейти к сле-

дующему разделу, описывающему функции для работы с датой и временем.

Если же необходимо управлять представлением даты и времени, то PostgreSQL позволяет это делать, хотя и с некоторыми ухищрениями.

Как уже говорилось, есть два независимых параметра, которыми можно управлять раздельно. Трудность в том, что оба они устанавливаются при помощи одной переменной DATESTYLE. Следует помнить, что управлять можно только внешним представлением данных. Внутри PostgreSQL хранит данные в формате, никак не связанном с их внешним представлением, предназначенным только для ввода и вывода. Формат команды psql таков:

```
SET DATESTYLE TO 'значение';
```

Регистр, как обычно для операторов SQL, не имеет значения.

Для определения порядка следования дня и месяца используйте значения «US» и «European» (табл. 4.3):

Таблица 4.3. Американский и европейский стили вывода даты

Значение	Смысл	Пример
US	Месяц, затем день	02/01/1997
European	День, затем месяц	01/02/1997

Для изменения формата отображения также используется DATESTYLE, но с другими значениями (табл. 4.4):

Таблица 4.4. Форматы отображения

Значение	Смысл	Пример
ISO	Стандарт ISO-8601, '-' для разделения полей	1997-02-01
SQL	Традиционный стиль	02/01/1997
Postgres	Собственный стиль	Sat Feb 01
German	Немецкий стиль	01.02.1997

В текущей версии формат Postgres ***по умолчанию отображает данные типа*** DATE ***в стиле*** SQL***, но для типа*** TIMESTAMP ***использует более длинный формат, как показано в примере.***

Переменной DATESTYLE присваивается пара значений, разделяемых запятой. Так, для того чтобы установить формат отображения SQL и европейский стиль записи (день перед месяцем), напишем:

```
SET DATESTYLE TO 'European, SQL';
```

Чтобы не устанавливать стиль представления даты в каждой сессии, можно задать его при установке или определить для сессии стиль по умолчанию.

Если требуется установить стиль по умолчанию для всей базы данных, то определите переменную окружения PGDATESTYLE до старта основного серверного процесса postmaster. Установка переменной окружения в UNIX, Linux и Cygwin выполняется одинаково:

```
PGDATESTYLE="European, SQL"
export PGDATESTYLE
```

Другой способ – передача параметра серверному процессу postmaster, но мы считаем, что переменные окружения удобнее.

Установить стиль даты для отдельного пользователя можно при помощи той же переменной окружения, но определенной для сессии, перед запуском psql. Локальные переменные имеют приоритет над глобальными.

Прежде чем продемонстрировать, как все это работает, познакомимся с функцией CAST, предназначенной для преобразования форматов. Хотя PostgreSQL тщательно следит за соответствием типов и, как правило, не возникает необходимости их явного преобразования, иногда эта функция бывает полезной. Вызов функции для преобразования в тип DATE:

```
CAST('string' TO DATE)
```

Для преобразования к типу, включающему время:

```
CAST('string' TO TIMESTAMP)
```

Нам также потребуется проделать небольшой трюк. Вероятно, в девяносто девяти случаях из ста вы будете выбирать данные из таблиц. Но в некоторых ситуациях можно использовать SELECT для получения данных, которые вообще отсутствуют в таблицах. Например, так:

```
bpsimple=# SELECT 'Fred';
?column?
-------------
 Fred
(1 row)

bpsimple=#
```

Получено предупреждение от PostgreSQL о том, что не указан ни один столбец, но оператор SELECT успешно выполнился и без указания таблицы, вернув заданную строку.

Это свойство можно использовать совместно с функцией CAST, чтобы выяснить, как PostgreSQL обращается с датой и временем, не создавая для своих экспериментов временных таблиц.

Попробуйте сами. Форматы дат

Мы начали работать, не устанавливая переменную окружения PGDATESTYLE, поэтому могли наблюдать поведение системы по умолчанию. Затем был установлен другой формат даты, и вы убедились, что это работает. Для даты установлен формат ISO, т. к. это всегда безопасный выбор, и вывод PostgreSQL осуществляется в том же формате. Нет никаких несоответствий в выражении и результате:

```
bpsimple=# SELECT CAST('1997-02-1' AS DATE);
?column?
-------------
1997-02-01
(1 row)

bpsimple=#
```

Используем стиль US и формат SQL и введем дату: (по-прежнему 1 февраля) в формате ISO, исключающем двусмысленность. Теперь выводимая дата имеет более привычный (для американцев), но двусмысленный формат MM/DD/YYYY:

```
bpsimple=# SET DateStyle TO 'US, SQL';
SET VARIABLE
bpsimple=# SELECT CAST('1997-02-1' AS DATE);
?column?
-------------
02/01/1997
(1 row)

bpsimple=#
```

Проверим, какое значение datestyle установлено в программе psql:

```
bpsimple=# SHOW datestyle;
NOTICE:  DateStyle is SQL with US (NonEuropean) conventions
SHOW VARIABLE

bpsimple=#
```

Теперь поработаем с другими форматами. Та же дата на входе, но для вывода устанавливается европейский стиль DD/MM/YYYY:

```
bpsimple=# SET DateStyle TO 'European, SQL';
SET VARIABLE
bpsimple=# SELECT CAST('1997-02-1' AS DATE);
  ?column?
-------------
 01/02/1997
(1 row)
bpsimple=#
```

Вернемся к формату ISO на входе и выходе. Значение European ни на что не влияет, поскольку перекрывается форматом ISO:

```
bpsimple=# SET DateStyle TO 'European, ISO';
SET VARIABLE
bpsimple=# SELECT CAST('1997-02-1' AS DATE);
  ?column?
-------------
 1997-02-01
(1 row)

bpsimple=#
```

Теперь используем тип TIMESTAMP, включающий время. Часы и минуты не указаны, поэтому по умолчанию имеют нулевые значения:

```
bpsimple=# SELECT CAST('1997-02-1' AS TIMESTAMP);
        ?column?
--------------------------
 1997-02-01 00:00:00+00
(1 row)

bpsimple=#
```

Здесь применяется однозначный и легче читаемый стиль PostgreSQL:

```
bpsimple=# SET DateStyle TO 'European, Postgres';
SET VARIABLE
bpsimple=# SELECT CAST('1997-02-1' AS TIMESTAMP);
           ?column?
-------------------------------
 Sat 01 Feb 00:00:00 1997 GMT
(1 row)

bpsimple=#
```

Как это работает

Как видите, можно менять способ отображения даты и времени, а также способ интерпретации неоднозначных входных значений, таких как '01/02/1997'.

С часовыми поясами все гораздо проще. Если в окружении правильно установлена переменная TZ, PostgreSQL учитывает ее значение без вмешательства извне.

Функции для работы с датой и временем

Теперь, выяснив, как обращаться с датой и временем, рассмотрим пару полезных функций, которые могут понадобиться при сравнении дат:

```
date_part(units required, value to use);
now()
```

Первая из них, date_part(), позволяет извлечь заданный компонент даты, например месяц.

Предположим, нужно выбрать из таблицы orderinfo строки, в которых дата размещения заказа приходится на сентябрь. Известно, что сентябрь – девятый месяц, поэтому можно написать:

```
bpsimple=# SELECT * FROM orderinfo WHERE date_part('month',date_placed)=9;
 orderinfo_id | customer_id | date_placed | date_shipped | shipping
--------------+-------------+-------------+--------------+----------
            3 |          15 | 2000-09-02  | 2000-09-12   |     3.99
            4 |          13 | 2000-09-03  | 2000-09-10   |     2.99
(2 rows)

bpsimple=#
```

Соответствующие записи получены. Заметьте, дата отображается в формате ISO. Компоненты, которые можно извлечь из данных типа DATE и TIMESTAMP, следующие:

- год – year
- месяц – month
- день – day
- час – hour
- минута – minute
- секунда – second

Можно сравнивать даты при помощи тех же операторов <>, <=, <, >, >=, =, которые применялись при сравнении чисел. Например:

```
bpsimple=# SELECT * FROM orderinfo WHERE date_placed >= CAST('2000-07-21' AS
bpsimple-# DATE);
 orderinfo_id | customer_id | date_placed | date_shipped | shipping
--------------+-------------+-------------+--------------+----------
            3 |          15 | 2000-09-02  | 2000-09-12   |     3.99
            4 |          13 | 2000-09-03  | 2000-09-10   |     2.99
            5 |           8 | 2000-07-21  | 2000-07-24   |     0.00
(3 rows)

bpsimple=#
```

Обратите внимание, что нам пришлось преобразовать строки в даты при помощи функции CAST и использовать однозначный стиль ISO.

Вторая функция, now(), просто возвращает текущие дату и время. Это может быть удобно, если, например, создается запись о заказе непосредственно во время телефонного звонка клиента или в режиме реального времени при работе в сети. Можно обратиться к now(), для того чтобы просто получить текущую дату:

```
bpsimple=# SELECT now();
              now
--------------------------
 2001-02-03 17:06:20+00
(1 row)

bpsimple=#
```

Над датами можно выполнять простые вычисления. Например, для того чтобы узнать, сколько дней прошло от даты размещения заказа до даты поставки, можно сформировать такой запрос:

```
bpsimple=# SELECT date_shipped - date_placed FROM orderinfo;
 ?column?
----------
        4
        1
       10
        7
        3
(5 rows)

bpsimple=#
```

Здесь возвращается количество дней между двумя датами, хранящимися в базе данных.

Связывание данных в таблицах

К этому моменту нам известно, как выбирать данные из таблиц, указывать, какие нам нужны столбцы и строки, как управлять представлением даты. Мы также узнали, как обращаться с датами в разных форматах.

Настало время заняться одной из наиболее важных тем языка SQL и реляционных баз данных вообще – автоматическим связыванием данных одной таблицы с данными другой. Хорошо то, что все это выполняется в операторе `SELECT` и все полученные ранее полезные сведения об этом операторе применимы и к работе с несколькими таблицами.

Связь двух таблиц

Прежде чем использовать в SQL несколько таблиц одновременно, повторим вкратце материал главы 2 о связях между таблицами.

Как вы помните, у нас есть таблица `customer`, в которой хранятся данные о наших клиентах, и таблица `orderinfo`, содержащая данные о сделанных ими заказах. Такая структура позволяет нам сохранять данные о клиенте один раз, независимо от количества сделанных им заказов. Две таблицы связаны вместе с помощью общего элемента данных `customer_id`, имеющегося в обеих таблицах.

Наглядно это можно представить как строку в таблице `customer`, содержащую значение `customer_id` и либо ни с чем не связанную, либо связанную с одной или несколькими строками таблицы `orderinfo`, содержащими такое же значение `customer_id` (рис. 4.1):

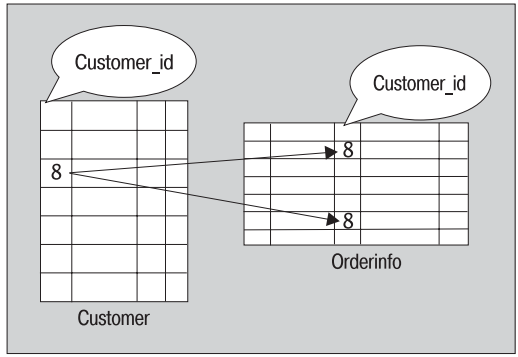


Рис. 4.1. Связь между таблицами customer и orderinfo

Можно сказать, что значение 8 в поле customer_id строки таблицы customer ссылается на две строки таблицы orderinfo, содержащие такие же значения customer_id. Разумеется, столбцы этих таблиц не обязаны называться одинаково, но, поскольку оба они содержат идентификатор клиента, было бы неразумно называть их разными именами.

Предположим, надо найти все заказы, сделанные нашим клиентом по имени Ann Stones. Логика подсказывает первое необходимое действие – найти этого клиента в таблице customer:

```
bpsimple=# SELECT customer_id FROM customer WHERE fname = 'Ann' AND lname =
bpsimple-# 'Stones';
customer_id
-------------
           8
(1 row)

bpsimple=#
```

Теперь идентификатор customer_id известен, и можно найти заказы:

```
bpsimple=# SELECT * FROM orderinfo WHERE customer_id = 8;
 orderinfo_id | customer_id | date_placed | date_shipped | shipping
--------------+-------------+-------------+--------------+--------------
            2 |           8 | 2000-06-23  | 2000-06-24   |     0.00
            5 |           8 | 2000-07-21  | 2000-07-24   |     0.00
(2 rows)

bpsimple=#
```

Такой способ работает, но состоит из двух шагов, между которыми надо помнить значение customer_id. В главе 2 уже говорилось о том, что SQL – это декларативный язык. Это значит, что мы должны сформулировать, что хотим получить в результате, а не описывать явно последовательность действий. Только что был продемонстрирован процедурный подход. Описаны два дискретных шага в решении задачи нахож-

дения заказов, сделанных одним из клиентов. Попробуем найти более изящное одношаговое решение.

Разумеется, с помощью SQL это можно сделать в один прием, написав, что нам нужны заказы, сделанные клиентом 'Ann Stones', и использовав тот факт, что информация хранится в таблицах `customer` и `orderinfo`, связанных при помощи столбца `customer_id`, имеющегося в обеих таблицах.

Новый элемент синтаксиса, который для этого применяется, – это расширенная инструкция `WHERE`:

```
SELECT <список столбцов> FROM <список таблиц> WHERE <условие объединения> AND
<условие выбора строк>
```

На самом деле это не так сложно, как кажется. Чтобы упростить первый пример, предположим, что нам уже известен идентификатор клиента (8), и выведем только даты заказов и имя клиента.

Нам надо указать столбцы, которые необходимо получить (имя клиента и дата размещения заказа), отразить тот факт, что таблицы связаны посредством поля `customer_id`, и указать, что нас интересуют только строки, в которых `customer_id = 8`.

Как видите, возникла небольшая трудность. Как выразить на SQL, какой из столбцов `customer_id` нам нужен – тот, который в таблице `customer`, или тот, что в `orderinfo`? Хотя и планировалось проверять их на равенство, в общем случае это может быть и не так; вопрос в том, как адресовать столбцы, встречающиеся более чем в одной таблице? Для этого используется расширенное имя столбца:

```
tablename.columnname
```

Теперь можно однозначно идентифицировать любой столбец базы данных. Вообще, PostgreSQL в ситуации, когда имя столбца в операторе `SELECT` входит только в одну таблицу, не требует явно указывать ее имя. В данном случае будем писать `customer.fname`, хотя достаточно было бы `fname`, поскольку это легче для понимания и полезно при изучении SQL. Таким образом, первая часть нашего оператора должна выглядеть так:

```
SELECT customer.fname, orderinfo.date_placed FROM customer, orderinfo;
```

Здесь сообщаем PostgreSQL, какие столбцы и таблицы собираемся использовать.

Теперь надо определить условия. У нас есть два разных условия: то, что `customer_id` равно 8, и то, что таблицы связаны, или объединены (joined), при помощи поля `customer_id`. Как и раньше при задании нескольких условий, указываем ключевое слово `AND`, поскольку каждое условие должно быть выполнено:

```
WHERE customer.customer_id = 8 AND customer.customer_id = orderinfo.customer_id;
```

Заметьте, что, ссылаясь на столбец customer_id, следует указывать его расширенное имя в форме tablename.columname несмотря на то, что не имеет значения, какое из них проверяется на равенство 8, т. к. уже написано, что их значения должны быть равны.

Объединив все вместе, получим:

```
bpsimple=# SELECT customer.fname, orderinfo.date_placed FROM customer,
bpsimple-# orderinfo WHERE customer.customer_id = 8 AND customer.customer_id
bpsimple-# =orderinfo.customer_id;
 fname | date_placed
-------+-------------
 Ann   | 06/23/2000
 Ann   | 07/21/2000
(2 rows)

bpsimple=#
```

Намного изящнее, чем пошаговое решение, не правда ли? Немаловажно и то, что, сформулировав запрос в одном операторе, мы позволяем PostgreSQL полностью оптимизировать процесс извлечения данных.

Теперь принцип известен, и мы попробуем решить исходную задачу: найти все заказы, сделанные клиентом Ann Stones, не зная при этом значения customer_id.

Попробуйте сами. Объединение таблиц

Так как теперь известно только имя, а не идентификатор клиента, SQL-выражение немного усложнится. Определим клиента по имени и получим ответ:

```
bpsimple=# SELECT customer.fname, orderinfo.date_placed FROM customer,
bpsimple-# orderinfo WHERE customer.fname = 'Ann' AND customer.lname =
bpsimple-# 'Stones' AND customer.customer_id = orderinfo.customer_id;
fname  | date_placed
-------+-------------
 Ann   | 2000-06-23
 Ann   | 2000-07-21
(2 rows)

bpsimple=#
```

Как это работает

Как и в предыдущем примере, указываем интересующие нас столбцы (customer.fname, orderinfo.date_placed), используемые таблицы (customer, orderinfo), условие выбора (customer.fname = 'Ann' AND customer.lname = 'Stones') и условие объединения таблиц (customer.customer_id = orderinfo.customer_id).

Язык SQL делает все остальное. Отметим, что количество заказов не имеет значения – их может не быть вовсе, а может быть один или не-

сколько, как в примере. Корректно составленный SQL-запрос успешно выполняется, даже если нет строк, удовлетворяющих поставленному условию.

Рассмотрим другой пример. Предположим, требуется вывести список всех имеющихся продуктов и значения их штрих-кодов. Названия товаров хранятся в таблице `items`, а значения кодов – в таблице `barcodes`. Связь между таблицами осуществляется по значениям столбцов `item_id`, имеющихся в обеих таблицах. Вы, вероятно, помните, что причиной разделения данных между таблицами было то, что многие товары имеют несколько штрих-кодов.

Благодаря приобретенным навыкам объединения таблиц мы знаем, что следует указать нужные нам столбцы, таблицы и то, как они связаны между собой. Можно даже упорядочить записи по себестоимости товаров.

Вводим SQL-оператор и получаем ответ:

```
bpsimple=# SELECT description, cost_price, barcode_ean FROM item, barcode
bpsimple-# WHERE barcode.item_id = item.item_id ORDER BY cost_price;
  description  | cost_price |  barcode_ean
---------------+------------+---------------
 Toothbrush    |       0.75 | 6241234586487
 Toothbrush    |       0.75 | 9473625532534
 Toothbrush    |       0.75 | 9473627464543
 Linux CD      |       1.99 | 6264537836173
 Linux CD      |       1.99 | 6241527746363
 Tissues       |       2.11 | 7465743843764
 Roman Coin    |       2.34 | 4587263646878
 Rubic Cube    |       7.45 | 6241574635234
 Picture Frame |       7.54 | 3453458677628
 Fan Small     |       9.23 | 6434564564544
 Fan Large     |      13.36 | 8476736836876
 Wood Puzzle   |      15.23 | 6241527836173
 Speakers      |      19.73 | 9879879837489
 Speakers      |      19.73 | 2239872376872
(14 rows)

bpsimple=#
```

Выглядит разумно, за исключением нескольких повторяющихся позиций – кажется, на складе у нас были колонки (speakers) только одного типа. Да и вообще, там нет такого количества разных товаров.

Что произошло?

Полученные нами знания по SQL позволяют подсчитать количество позиций:

```
bpsimple=# SELECT * FROM item;
```

В результате будет выбрано 11 строк.

У нас имеется 11 наименований товаров, а предыдущий запрос вернул 14 строк. Мы где-то ошиблись? Нет, дело в том, что некоторые товары, например зубная щетка (Toothbrush), имеют по несколько штрих-кодов. Поэтому PostgreSQL просто повторила название товара для каждого нового штрих-кода, т. к. она должна вывести все коды и соответствующие им названия товаров.

В этом можно убедиться, добавив в запрос идентификатор товара:

```
bpsimple=# SELECT item.item_id, description, cost_price, barcode_ean FROM
bpsimple-# item, barcode WHERE barcode.item_id = item.item_id ORDER BY
bpsimple-# cost_price;
item_id |  description  | cost_price |  barcode_ean
--------+---------------+------------+---------------
      8 | Toothbrush    |       0.75 | 6241234586487
      8 | Toothbrush    |       0.75 | 9473625532534
      8 | Toothbrush    |       0.75 | 9473627464543
      3 | Linux CD      |       1.99 | 6264537836173
      3 | Linux CD      |       1.99 | 6241527746363
      4 | Tissues       |       2.11 | 7465743843764
      9 | Roman Coin    |       2.34 | 4587263646878
      2 | Rubic Cube    |       7.45 | 6241574635234
      5 | Picture Frame |       7.54 | 3453458677628
      6 | Fan Small     |       9.23 | 6434564564544
      7 | Fan Large     |      13.36 | 8476736836876
      1 | Wood Puzzle   |      15.23 | 6241527836173
     11 | Speakers      |      19.73 | 9879879837489
     11 | Speakers      |      19.73 | 2239872376872
(14 rows)

bpsimple=#
```

Обратите внимание, что было указано, из какой таблицы берется поле item_id, т. к. столбцы с таким именем имеются в обеих таблицах – item и barcode.

Теперь понятно, что произошло. Если данные, возвращаемые оператором SELECT, выглядят немного странно, хорошим способом разобраться в том, что происходит, может стать добавление в запрос столбцов с идентификаторами.

Использование псевдонимов для таблиц

Вспомните, ранее в этой главе мы узнали, как изменить название столбца при выводе на более удобное с помощью инструкции AS. При желании можно заменить и названия таблиц. Это бывает удобно в тех редких случаях, когда приходится давать два имени одной таблице, но чаще это используется для сокращения записи. Мы увидим, что этот способ часто применяется в графических утилитах для облегчения генерации SQL-выражений.

Для того чтобы создать псевдоним, просто поместите его непосредственно после имени таблицы в инструкции FROM. После этого псевдоним сможет заменить настоящее имя в любой части SQL-оператора.

Проще объяснить это на примере. Предположим, у нас есть простой оператор SQL:

```
SELECT lname FROM customer;
```

Как мы уже видели, в названии столбца всегда можно указывать явное имя таблицы:

```
SELECT customer.lname FROM customer;
```

Если указать для таблицы псевдоним cu, то и в качестве префикса может выступать cu:

```
SELECT cu.lname FROM customer cu;
```

Здесь cu добавлено после имени таблицы и оно было использовано в имени столбца.

Замена имени для единственной таблицы не имеет большого смысла. Значительно больший толк из этого можно извлечь при работе с несколькими таблицами.

Вернемся к предыдущему примеру:

```
SELECT customer.fname, orderinfo.date_placed FROM customer, orderinfo WHERE
customer.fname = 'Ann' AND customer.lname = 'Stones' AND customer.customer_id =
orderinfo.customer_id;
```

С применением псевдонимов он выглядит так:

```
SELECT c.fname, o.date_placed FROM customer c, orderinfo o WHERE c.fname =
'Ann' AND c.lname = 'Stones' AND c.customer_id = o.customer_id;
```

Объединение трех таблиц

Научившись объединять две таблицы, можем ли мы применить этот метод к трем и более таблицам? Разумеется, да. Язык SQL очень логичен и, если можно выполнить некоторое действие над N объектами, почти всегда можно выполнить его и над N+1 объектами. Конечно, чем больше таблиц входит в запрос, тем большую работу приходится выполнять PostgreSQL, поэтому выборки по многим таблицам могут выполняться довольно медленно, особенно если некоторые из таблиц имеют большой размер.

Предположим, требуется установить связь между клиентами и идентификаторами заказанных ими товаров.

На схеме данных видно, что переход от клиента к заказанным товарам требует использования трех таблиц. Нам потребуются customer, orderinfo и orderline (рис. 4.2):

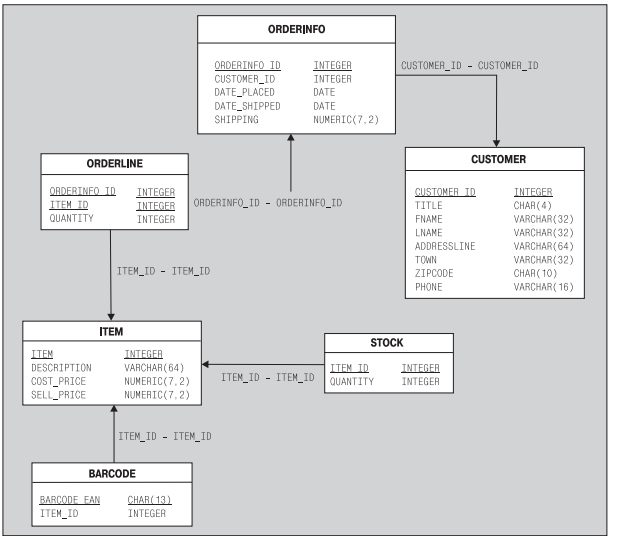


Рис. 4.2. Отношения между таблицами базы данных

Если перерисовать предыдущую диаграмму для трех таблиц, получится то, что показано на рис. 4.3:

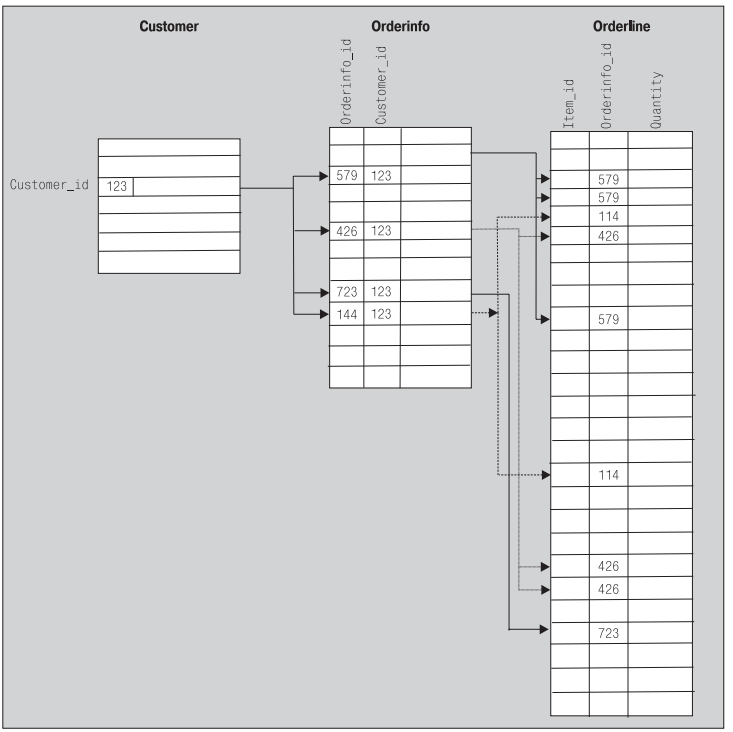


Рис. 4.3. Отношения между таблицами customer, orderinfo и orderline

Здесь видно, что клиенту '123' соответствуют несколько строк таблицы orderinfo, содержащие значения orderinfo_id '579', '426', '723' и '114', каждое из которых, в свою очередь, ссылается на одну или несколько записей в таблице orderline.

Заметьте, между таблицами customer и orderline нет прямой связи. Следует использовать таблицу orderinfo, т. к. она содержит информацию, связывающую клиентов с их заказами.

Попробуйте сами. Объединение трех таблиц

Создадим объединение из трех таблиц, чтобы узнать, какие товары в действительности заказывала клиент Ann Stones.

Начнем со столбцов:

```
SELECT customer.fname, customer.lname, orderinfo.date_placed,
orderline.item_id, orderline.quantity
```

Затем перечислим используемые таблицы:

```
FROM customer, orderinfo, orderline
```

Теперь определим условие объединения таблиц orderinfo и customer:

```
WHERE customer.customer_id = orderinfo.customer_id
```

Также требуется определить условие объединения таблиц orderinfo и orderline:

```
orderinfo.orderinfo_id = orderline.orderinfo_id
```

И наконец, условие:

```
customer.fname = 'Ann' AND customer.lname = 'Stones';
```

Собрав все вместе, получим:

```
bpsimple=# SELECT customer.fname, customer.lname, orderinfo.date_placed,
bpsimple-# orderline.item_id,orderline.quantity
bpsimple-# FROM customer, orderinfo, orderline
bpsimple-# WHERE
bpsimple-# customer.customer_id = orderinfo.customer_id AND
bpsimple-# orderinfo.orderinfo_id = orderline.orderinfo_id AND
bpsimple-# customer.fname = 'Ann' AND
bpsimple-# customer.lname = 'Stones';
fname | lname  | date_placed | item_id | quantity
------+--------+-------------+---------+----------
 Ann  | Stones | 2000-06-23  |       1 |        1
 Ann  | Stones | 2000-06-23  |       4 |        2
 Ann  | Stones | 2000-06-23  |       7 |        2
 Ann  | Stones | 2000-06-23  |      10 |        1
 Ann  | Stones | 2000-07-21  |       1 |        1
```

```
 Ann  | Stones | 2000-07-21  |       3 |        1
(6 rows)

bpsimple =#
```

Заметьте, что дополнительные пробелы и разделители строк добавлены только для удобства чтения, для SQL они не обязательны. Только получив символ «;», программа `psql` начинает интерпретацию введенного текста.

Попробуйте сами. Объединение четырех таблиц

Видя, как легко оказалось перейти от двух к трем таблицам, сделаем еще один шаг и выведем описания всех товаров, заказанных Ann Stones. Для этого нам потребуется еще одна таблица, `item`, в которой хранятся описания. Остальная часть запроса практически не изменилась:

```
bpsimple=# SELECT customer.fname, customer.lname, orderinfo.date_placed,
bpsimple-# item.description, quantity
bpsimple-# FROM customer, orderinfo, orderline, item
bpsimple-# WHERE
bpsimple-# customer.customer_id = orderinfo.customer_id AND
bpsimple-# orderinfo.orderinfo_id = orderline.orderinfo_id AND
bpsimple-# orderline.item_id = item.item_id AND
bpsimple-# customer.fname = 'Ann' AND
bpsimple-# customer.lname = 'Stones';
fname | lname  | date_placed | description | quantity
------+--------+-------------+-------------+----------
Ann   | Stones | 2000-06-23  | Wood Puzzle |        1
Ann   | Stones | 2000-07-21  | Wood Puzzle |        1
Ann   | Stones | 2000-07-21  | Linux CD    |        1
Ann   | Stones | 2000-06-23  | Tissues     |        2
Ann   | Stones | 2000-06-23  | Fan Large   |        2
Ann   | Stones | 2000-06-23  | Carrier Bag |        1
(6 rows)

bpsimple=  #
```

Как это работает

Тот, кто понял, как работает объединение трех таблиц, без труда обобщит эту идею на большее их количество. Мы добавили описание товара к списку отображаемых столбцов, добавили таблицу `item` к списку используемых таблиц и условие объединения таблицы `item` с уже имеющимися таблицами, `orderline.item_id = item.item_id`.

Как видите, головоломка Wood Puzzle встречается дважды, т. к. на нее было два разных заказа.

В этом операторе `SELECT` выведено как минимум по одному столбцу из каждой таблицы объединения. На самом деле это не обязательно: если нас интересует только имя клиента и описание товара, можно просто не указывать остальные поля.

Версия с меньшим количеством выбираемых столбцов также корректна и, возможно, работает немного быстрее:

```
SELECT customer.fname, customer.lname, item.description
FROM customer, orderinfo, orderline, item
WHERE
    customer.customer_id = orderinfo.customer_id AND
    orderinfo.orderinfo_id = orderline.orderinfo_id AND
    orderline.item_id = item.item_id AND
    customer.fname = 'Ann' AND
    customer.lname = 'Stones';
```

В последнем примере этой главы вернемся к тому, что уже делали раньше: удалим повторяющиеся строки, используя ключевое слово DISTINCT.

Попробуйте сами. Использование дополнительных условий

Предположим, мы решили узнать, какие виды товаров покупает Ann Stones. Все что надо вывести – это упорядоченный список описаний товаров. Не требуется выводить имя клиента, поскольку оно известно (и используется в условии выбора данных). Надо вывести только столбец item.description с использованием параметра DISTINCT, чтобы каждая позиция присутствовала в списке только один раз, даже если соответствующий товар был приобретен дважды.

```
bpsimple=# SELECT DISTINCT  item.description
bpsimple-# FROM customer, orderinfo, orderline, item
bpsimple-# WHERE
bpsimple-# customer.customer_id = orderinfo.customer_id AND
bpsimple-# orderinfo.orderinfo_id = orderline.orderinfo_id AND
bpsimple-# orderline.item_id = item.item_id AND
bpsimple-# customer.fname = 'Ann' AND
bpsimple-# customer.lname = 'Stones'
bpsimple-# ORDER BY item.description;
description
------------
Carrier Bag
Fan Large
Linux CD
Tissues
Wood Puzzle
(5 rows)

bpsimple=#
```

Как это работает

Просто взят предыдущий SQL-оператор, удалены ненужные столбцы, добавлено ключевое слово DISTINCT после SELECT, чтобы исключить повторения, и добавлено ORDER BY после инструкции WHERE.

Это одно из достоинств SQL: изучив какую-либо возможность, можно применять ее и в более общих случаях. К примеру, ORDER BY работает с несколькими таблицами точно так же, как и с одной.

Резюме

Это была довольно длинная глава, но мы узнали много нового.

Был изучен оператор SELECT. Представлены такие детали, как выбор столбцов и строк, упорядочение вывода и подавление вывода дублирующих строк. Также затронуты типы данных, способы настройки форматов ввода и вывода дат и применение дат в условных выражениях.

Затем мы перешли к одной из главных возможностей SQL – его способности объединять таблицы. После первого удачного опыта объединения двух таблиц было показано, как легко этот метод может быть расширен до трех и даже до четырех таблиц. В заключении приобретенные ранее знания были использованы для удаления лишних столбцов и повторяющихся строк в операторе выборки из четырех таблиц.

Приятно констатировать, что состоялось знакомство со всеми каждодневно используемыми возможностями оператора SELECT, а если вы поняли, как работает SELECT, то освоение оставшейся части языка SQL не вызовет затруднений. Мы вернемся к оператору SELECT в главе 7, чтобы рассмотреть некоторые дополнительные возможности, но большая часть его средств, применяемых в повседневной работе, была рассмотрена в этой главе.

5

Графические программы для работы с PostgreSQL

Обычно для создания и администрирования базы данных PostgreSQL применяется утилита командной строки `psql`, с которой мы уже работали в предыдущих главах. Программы с командной строкой, аналогичные `psql`, практически всегда используются в коммерческих базах данных. Это, например SQL*Plus, поставляемая с Oracle. Несмотря на то что утилиты командной строки полнофункциональны – в том смысле, что позволяют выполнить все необходимые действия, нельзя сказать, что они очень удобны в работе. С другой стороны, они не предъявляют больших требований к аппаратным средствам: графической подсистеме, памяти и т. д.

В этой главе рассмотрим несколько альтернативных `psql` вариантов доступа к PostgreSQL. Некоторые из программ можно применять и для администрирования баз данных. Специальный пользователь, отвечающий за поддержку СУБД, выполняет такие административные задачи, как создание новых пользователей, назначение прав доступа, оптимизация базы данных. Об администрировании PostgreSQL поговорим в главе 11. Здесь же обратимся к тем функциям, которые доступны обычным пользователям.

Начнем с краткого обзора команд `psql`, а затем обратимся к следующим темам:

- `psql`
- ODBC
- `pgAdmin`
- `Kpsql`

- PgAccess
- Microsoft Access
- Microsoft Excel

psql

Программа psql позволяет устанавливать соединение с базой данных, выполнять запросы и административные действия: создание базы данных, добавление новых таблиц, ввод и модификацию данных, исполнение SQL-команд.

Запуск psql

Формат команды psql таков:

```
psql [параметры] [имя БД [имя пользователя]]
```

Как вы уже видели, при запуске psql указывается имя базы данных, с которой должно быть установлено соединение. Для чего необходимо знать имя сервера и номер порта, а также имя пользователя и пароль. По умолчанию соединение устанавливается с базой данных на локальной машине с именем, совпадающим с именем пользователя.

Можно соединиться с указанной базой данных, задав ее имя при запуске psql:

```
$ psql -d bpsimple
```

Умолчания для имен базы данных, пользователя, сервера и номера порта могут быть изменены путем определения переменных окружения PGDATABASE, PGUSER, PGHOST и PGPORT соответственно.

Значения параметров –d, -U, -h и –p в командной строке psql также отменяют соответствующие значения по умолчанию.

Программа** psql **может запуститься только в случае успешного соединения с базой данных. Вследствие этого возникает проблема «курицы и яйца» при создании первой базы данных: необходим пользователь и база данных для установления первоначального соединения. В этом качестве выступает пользователь по умолчанию с именем** postgres**, созданный в процессе установки (глава 3). Также при установке PostgreSQL всегда создается база данных** template1**. Соединившись с ней, можно создать новую базу данных и либо перезапустить программу** psql**, либо выполнить ее внутреннюю команду** \c **для соединения с новой базой данных.

Полный список параметров psql можно получить с помощью команды:

```
$ psql --help
```

Команды psql

При запуске psql попытается прочитать файл конфигурации .psqlrc, если он имеется в домашнем каталоге текущего пользователя и доступен для чтения. Этот файл похож на сценарий конфигурации командного интерпретатора и может содержать команды psql, определяющие такие параметры, как формат вывода таблиц и им подобные. От чтения конфигурационного файла можно отказаться, запустив psql с параметром -X.

Запустившись, psql выводит приглашение на ввод команд, состоящее из имени базы данных, с которой установлено соединение, и символов =#. В некоторых случаях это может быть =>. Команды могут быть двух типов: внутренние и SQL-команды. Можно ввести любой оператор SQL, поддерживаемый PostgreSQL, и psql выполнит его.

> ***Список всех поддерживаемых SQL-команд можно получить при помощи внутренней команды*** \h ***и подсказку по определенной команде:*** \h <команда SQL>. ***Внутренняя команда*** \? ***выводит список всех внутренних команд.***

Команды psql могут занимать несколько строчек, в таких случаях приглашение изменяется на -# или ->, показывая, что программа ожидает продолжения ввода:

```
$ psql -d bpsimple
...
bpsimple=#SELECT *
bpsimple-#FROM customer
bpsimple-#;
...
$
```

Можно сообщить psql, что ввод многострочного оператора SQL закончен, для чего необходимо поставить в конце команды точку с запятой. Имейте в виду, что точка с запятой служит знаком окончания ввода, а не является частью SQL-команды. В предыдущем примере можно было, например, написать инструкцию WHERE на следующей строке.

Запустив программу psql с параметром -S, можно отказаться от деления вводимых команд на несколько строк, в этом случае не потребуется добавлять точку с запятой в конце команды. Приглашение psql будет иметь вид ^>, напоминая о том, что действует однострочный режим.

Внутренние команды psql предназначены для выполнения операций, явно не поддерживаемых языком SQL, таких как вывод списка всех доступных таблиц и выполнение сценариев. Все внутренние команды начинаются с обратной косой черты и не допускают переноса на следующую строку.

История команд

Каждая команда, выполненная программой psql, записывается в историю команд, откуда она может быть извлечена для повторного выполнения или редактирования. Для просмотра списка команд и их редактирования применяются клавиши со стрелками. Данная возможность доступна всегда, если только не отключена параметром –n при запуске (или не исключена при компиляции). Историю запросов можно вывести командой \s или сохранить в файле: \s <файл>.

Последний выполненный запрос хранится в буфере запросов. Его содержимое можно посмотреть с помощью команды \p, а команда \r очищает буфер. Команда \e позволяет редактировать содержимое буфера во внешнем редакторе. По умолчанию вызывается редактор vi (по крайней мере, в Linux и UNIX), но вы можете указать свой собственный любимый редактор, определив переменную окружения EDITOR перед запуском psql.

Сценарии psql

Можно объединить несколько команд psql (как SQL-команд, так и внутренних) в файле и использовать его в качестве простого сценария. Внутренняя команда \i выполняет чтение команд psql из файла.

Эта возможность особенно полезна при создании и заполнении таблиц, она уже применялась для создания учебной базы данных bpsimple. Вот часть использованного нами файла сценария create_tables:

```
create table customer
(
    customer_id             serial                      ,
    title                   char(4)                     ,
    fname                   varchar(32)                 ,
    lname                   varchar(32)          not null,
    addressline             varchar(64)                 ,
    town                    varchar(64)                 ,
    zipcode                 char(10)             not null,
    phone                   varchar(16)                 ,
    CONSTRAINT              customer_pk PRIMARY KEY(customer_id)
);

create table item
(
    item_id                         serial                      ,
    description                     varchar(64)          not null,
    cost_price                      numeric(7,2)                ,
    sell_price                      numeric(7,2)                ,
    CONSTRAINT                      item_pk PRIMARY KEY(item_id)
);
```

Файлу дано общепринятое расширение .sql и запустили его внутренней командой \i:

```
bpsimple=#\i create_tables.sql
CREATE TABLE
CREATE TABLE
…
bpsimple=#
```

Другим вариантом применения файлов сценариев может быть создание простых отчетов. Чтобы следить за ростом нашей базы данных, можно поместить несколько команд в файл и запускать его время от времени. Чтобы узнать количество клиентов и сделанных ими заказов, создайте файл сценария report.sql, поместите туда указанные строки и выполните его в сессии psql:

```
select count(*) from customer;
select count(*) from orderinfo;
```

Другим способом является запуск psql с параметром -f, программа выполнит указанные в файле команды и завершится:

```
$ psql -f report.sql bpsimple
  count
---------
     15
(1 row)

  count
---------
      5
(1 row)
$
```

Вывод запроса может быть перенаправлен в файл, указанный параметром -o в командной строке, либо в файл или программу-фильтр при помощи внутренней команды \o непосредственно из сессии.

Исследование базы данных

Структуру базы данных можно изучать с помощью нескольких встроенных команд psql.

Команда \d выводит список всех отношений (то есть таблиц, последовательностей и представлений, если они имеются) базы данных. Команда \dt ограничивается перечислением таблиц.

Подробное описание любой таблицы можно получить, выполнив команду \d имятаблицы:

```
bpsimple=# \d customer
                        Table "customer"
  Attribute  |         Type          |         Modifier
-------------+-----------------------+----------------------------------
```

```
customer_id | integer               | not null default nextval(...)
title       | character(4)          |
fname       | character varying(32) |
lname       | character varying(32) | not null
addressline | character varying(64) |
town        | character varying(32) |
zipcode     | character(10)         | not null
phone       | character varying(16) |
Index: customer_pk

bpsimple=#
```

Более подробные сведения о внутренних командах `psql` можно найти в таблицах следующего раздела, а также в оперативной справке по PostgreSQL.

Краткий справочник по параметрам командной строки

Параметры командной строки `psql` и их значения приведены в табл. 5.1:

Таблица 5.1. Параметры psql

Параметр	Значение
`-a`	Включить эхо входных данных.
`-A`	Режим вывода таблиц без выравнивания (аналогично `-P format=unaligned`).
`-c <query>`	Выполнить единственный запрос (или внутреннюю команду) и завершить выполнение.
`-d <dbname>`	Определяет базу данных (по умолчанию `$PGDATABASE` или имя пользователя).
`-e`	Отображать запросы, отправленные на сервер базы данных.
`-E`	Отображать запросы, генерируемые внутренними командами.
`-f <filename>`	Выполнить запросы из файла и завершить выполнение.
`-F <string>`	Установить разделитель полей (по умолчанию «|») (аналогично `-P fieldsep=<string>`).
`-h <host>`	Определяет сервер базы данных (по умолчанию `$PGHOST` или локальная машина).
`-H`	Режим вывода таблиц в формате HTML (аналогично `-P format=html`).
`-l`	Вывести список доступных баз данных и завершить работу.
`-n`	Запретить редактирование строки.
`-o <filename>`	Вывести результат запроса в файл. Используйте `|pipe` для перенаправления вывода в программу-фильтр.

Таблица 5.1. (продолжение)

Параметр	Значение
-p <port>	Определяет порт сервера базы данных (по умолчанию $PGPORT или предопределенное при компиляции значение, обычно 5432).
-P var[=arg]	Установить параметр печати var в значение arg (см. команду \pset).
-q	Отключение вывода сообщения (только вывод результата запросов).
-R <string>	Установить разделитель записей (по умолчанию – разделитель строк) (аналогично -P recordsep=<строка>).
-s	Пошаговый режим (с подтверждением каждого запроса).
-S	Однострочный режим (запрос заканчивается разделителем строк, а не точкой с запятой).
-t	Выводить только строки (аналогично -P tuples_only).
-T <text>	Установить параметры (width, border) тега таблицы HTML (аналогично -P tableattr=<текст>).
-U <username>	Указать имя пользователя (по умолчанию $PGUSER или текущий пользователь).
-v name=val	Присвоить psql-переменной name значение val.
-V	Сообщить версию и завершить работу.
-W	Запросить пароль (выполняется автоматически, если пароль необходим).
-x	Включить развернутый вывод таблиц (аналогично -P expanded).
-X	Не использовать конфигурационный файл (~/.psqlrc).

Краткий справочник по внутренним командам

Внутренние команды, поддерживаемые psql, представлены в табл. 5.2:

Таблица 5.2. Внутренние команды psql

Команда	Значение
\a	Переключение между режимами с выравниванием и без выравнивания.
\c[onnect] [dbname|-[user]]	Установить новое соединение. Используйте символ - (дефис) в качестве имени БД, если необходимо соединиться с базой данных по умолчанию с указанием имени пользователя.
\C <title>	Установить заголовок таблицы при выводе (аналогично \pset title).
\copy ...	Выполнить SQL-команду COPY на клиентской машине.
\copyright	Вывести соглашение об условиях использования и распространения PostgreSQL.

Команда	Значение
\d <table>	Вывести описание таблицы (либо представления, индекса, последовательности).
\d{t|i|s|v}	Список таблиц/индексов/последовательностей/представлений.
\d{p|S|l}	Список прав доступа/системных таблиц/больших объектов
\da	Список агрегированных функций
\dd [object]	Список комментариев к таблице, типу, функции, оператору.
\df	Список функций.
\do	Список операторов.
\dT	Список типов данных.
\e [file]	Редактировать текущий буфер запроса или файл во внешнем редакторе.
\echo <text>	Вывести текст в стандартный поток вывода (stdout).
\encoding <encoding>	Установить кодировку клиента.
\f <sep>	Изменить разделитель полей.
\g [file]	Послать запрос на сервер базы данных (а результаты – в file или |pipe).
\h [cmd]	Подсказка по синтаксису SQL-команд, * для описания всех команд.
\H	Включить режим вывода в формате HTML.
\i <file>	Ввести и выполнить запрос из файла.
\l	Список всех баз данных.
\lo_export, \lo_import, \lo_list, \lo_unlink	Операции с большими объектами.
\o [file]	Направить результаты запросов в file или |pipe (в файл или в конвейер).
\p	Показать содержимое текущего буфера запросов.
\pset <opt>	Установить параметр вывода таблиц; возможные значения: format, border, expanded, fieldsep, null, recordsep, tuples_only, title, tableattr, pager.
\q	Завершить работу с psql.
\qecho <text>	Вывести текст в буфер запроса (см. \o).
\r	Очистить буфер запроса.
\s [file]	Вывести или сохранить в файле [file] историю команд.
\set <var> <value>	Установить внутреннюю переменную.

Таблица 5.2. (продолжение)

Команда	Значение
\t	Показывать только строки (переключатель режимов).
\T <tags>	Теги таблиц HTML.
\unset <var>	Удалить внутреннюю переменную.
\w <file>	Вывести текущий буфер запроса в файл <file>.
\x	Переключатель расширенного вывода.
\z	Вывести список прав доступа к таблице.
\! [cmd]	Запустить командный интерпретатор или выполнить команду.

ODBC

Некоторые из описанных в этой главе программ используют стандартный интерфейс ODBC для соединения с PostgreSQL. Протокол ODBC определяет общий интерфейс для баз данных и базируется на программных интерфейсах X/Open и ISO/IEC. Аббревиатура ODBC расшифровывается как «открытый интерфейс доступа к базам данных» (Open DataBase Connectivity) и не ограничивается (как это часто считается) клиентами под Microsoft Windows. Для того чтобы использовать ODBC на какой-либо клиентской машине, нам потребуется приложение, написанное с применением интерфейса ODBC, и драйвер для соответствующей базы данных.

СУБД PostgreSQL поставляется с ODBC-драйвером `psqlodbc`, входящим в дистрибутив с исходными текстами, который при желании может быть откомпилирован и установлен. Обычно клиентские приложения выполняются на разных машинах, в общем случае имеющих различную архитектуру. Поэтому нам может понадобиться откомпилировать драйверы ODBC также и на платформах клиентов. Например, сервер базы данных может работать на UNIX или Linux, а приложения клиентов могут выполняться в Windows.

К счастью, если клиент запускается под Microsoft Windows, то уже откомпилированный ODBC-драйвер можно получить с сайта Great Bridge.

- `psqlodbc-registry.zip`
- `psqlodbc-07_01_0003.zip`

Оба этих файла или их более свежие версии находятся по адресу *http://www.greatbridge.org/project/pgadmin*.

В Microsoft Windows драйверы ODBC регистрируются в системе при помощи соответствующего апплета Панели управления (рис. 5.1):

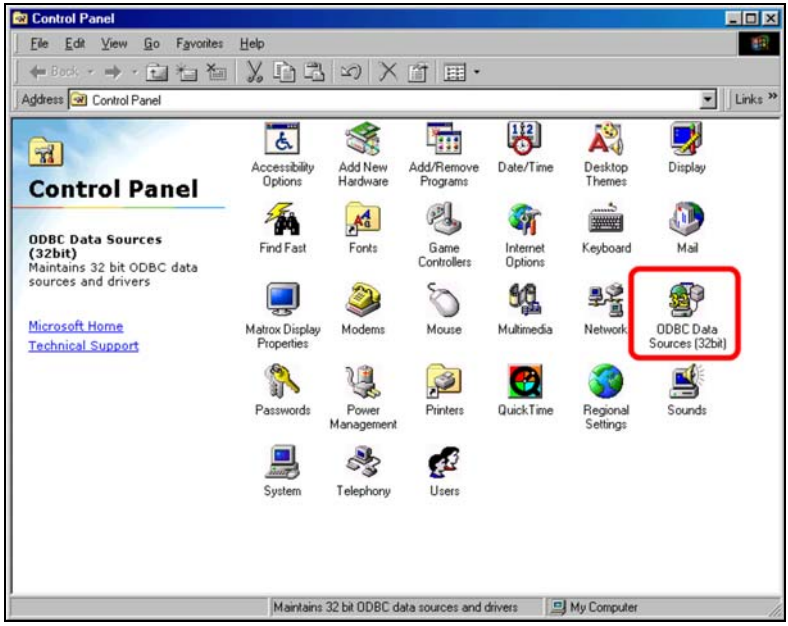


Рис. 5.1. Компоненты Панели управления Windows

На первой вкладке окна этого апплета показаны установленные драйверы ODBC (рис. 5.2):

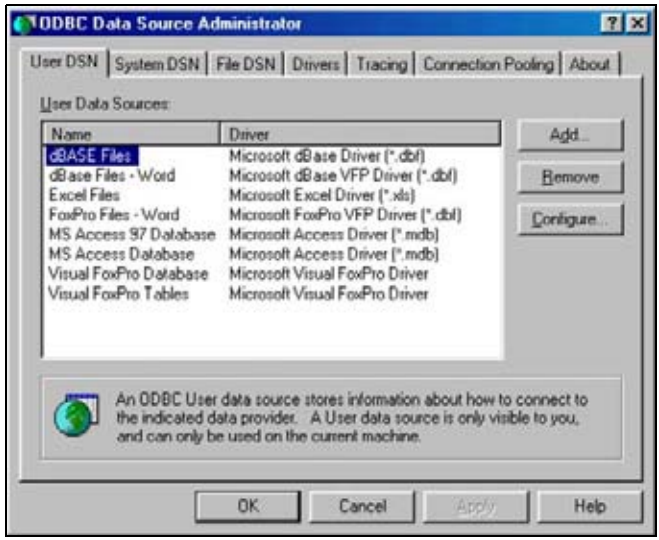


Рис. 5.2. Администратор источников данных ODBC

Процедура установки драйвера ODBC для PostgreSQL состоит из двух шагов. Во-первых, надо скопировать исполняемый модуль (в виде DLL-файла Windows) в соответствующее место. Во-вторых, необходимо зарегистрировать этот драйвер, чтобы ODBC знал о его присутствии в системе.

Файл драйвера в формате DLL называется psqlodbc.dll и находится в архиве `psqlodbc-07_01_0003.zip`. Его необходимо скопировать в подкаталог `SYSTEM` того каталога, куда установлена операционная система, обычно это `C:\WINDOWS\SYSTEM` для Windows 9x/ME или `C:\WINNT\SYSTEM32` для Windows NT/2000.

Можно зарегистрировать драйвер, создав соответствующие параметры в реестре вручную либо запустив файл `psqlodbc.reg`, находящийся в `psqlodbc-registry.zip`. В результате выполнения файла в реестре будут созданы необходимые записи.

> ***В предыдущих версиях ODBC драйвер для PostgreSQL входил в самоустанавливающийся исполняемый файл*** `postdrv.exe`***. Эта программа выполняла как установку, так и регистрацию, но драйвер этой версии не совместим с PostgreSQL версий старше 6.4. Проверьте, нет ли на сайте Great Bridge более свежей версии, если предпочитаете работать с самоустанавливающимся дистрибутивом.***

Выполнив оба этих шага, можно убедиться в том, что драйвер успешно установлен, выбрав вкладку Drivers в окне ODBC Data Source Administrator (Администрирование источников данных ODBC). PostgreSQL теперь присутствует в списке (рис. 5.3):

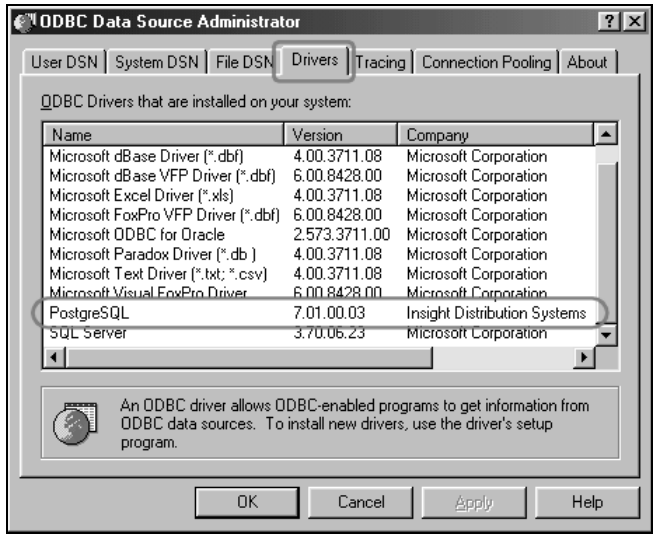


Рис. 5.3. Драйвер PostgreSQL установлен

Теперь можно использовать ODBC-приложения для работы с PostgreSQL. Чтобы получить доступ к определенной базе данных, надо создать соответствующий «источник данных» (data source). Для этого на вкладке User DSN окна ODBC Data Source Administrator создайте источник данных, который будет доступен текущему пользователю. Выбрав же вкладку System DSN, можно создать источник данных, доступный всем пользователям. Нажав кнопку Add, вы в диалоговом окне сможете выбрать драйвер для своего источника данных (рис. 5.4):

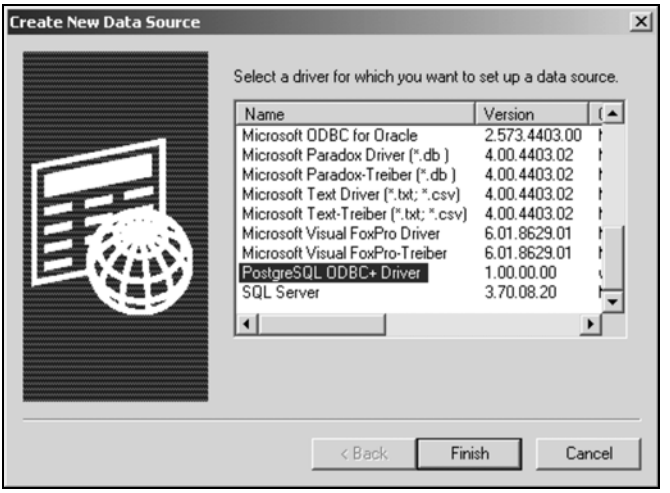


Рис. 5.4. Создание нового источника данных

Нажмите кнопку Finish.

Теперь у нас есть запись о драйвере PostgreSQL, который необходимо сконфигурировать. Диалоговое окно PostgreSQL Driver Setup предназначено для ввода параметров созданного источника данных (рис. 5.5):

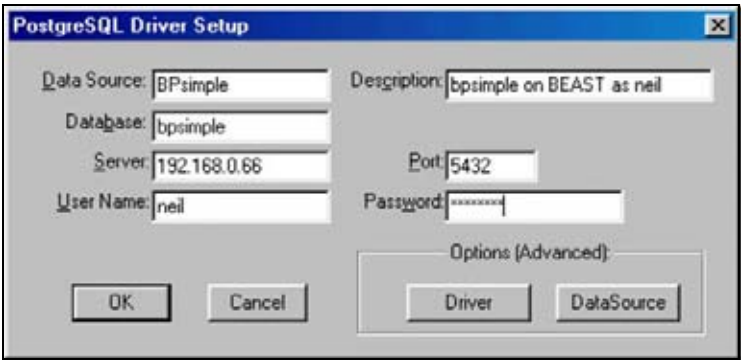


Рис. 5.5. Настройка драйвера

В поле Data Source укажем имя источника данных, в поле Description введем его описание и укажем сетевые настройки. Здесь введем IP-адрес сервера Linux. Если в сети работает сервер имен, такой как DNS или WINS, можно указать имя сервера. Укажем также имя пользователя и пароль, которые будут использоваться при соединении с выбранным сервером базы данных.

Нажав кнопки Driver и Data Source, можно задать дополнительные параметры. Если вам необходимо редактировать записи в базе данных с помощью приложения ODBC, убедитесь, что выключен режим «Read Only» (только чтение).

Теперь у нас есть возможность доступа к PostgreSQL из ODBC-приложений. В этой главе коротко рассмотрим две таких программы. Позже будет показано, как создавать клиентские приложения, работающие с PostgreSQL, с помощью Microsoft Access, но сначала познакомимся с программой pgAdmin.

pgAdmin

Программа pgAdmin предоставляет полноценный графический интерфейс к базам данных PostgreSQL. Эта программа поддерживается и свободно распространяется Great Bridge по общедоступной лицензии GNU GPL.

Как указано на сайте Great Bridge, «pgAdmin представляет собой программу общего назначения для создания, сопровождения и администрирования баз данных PostgreSQL. Она работает под Windows 95/98/ME и NT/2000.»

Основные ее возможности:

- Создание запросов произвольного вида с помощью SQL Wizard
- Встраиваемые (plug-in) экспортные модули, позволяющие экспортировать результаты SQL-запросов
- Средства просмотра и создания баз данных, таблиц, индексов, последовательностей, представлений, триггеров, функций и языков
- Диалог настройки параметров пользователей, групп и привилегий
- Слежение за версиями и генерация сценариев обновления
- Мастер импорта данных
- Мастер миграции базы данных (Database Migration Wizard)
- Встроенные (стандартные) отчеты по базам данных, таблицам, индексам, последовательностям, языкам и представлениям
- Возможность добавления пользовательских отчетов (при наличии Seagate Crystal Reports)

В этой главе не хватило бы места для описания всех функций программы pgAdmin, поэтому рассмотрим только процедуры установки и запуска этого многофункционального инструмента.

Предварительно потребуется установить ODBC-драйвер psqlodbc для PostgreSQL на той клиентской машине, где предполагается произвести установку pgAdmin. Подробности этого процесса рассмотрены в разделе «ODBC» данной главы.

Еще одно требование для запуска pgAdmin – наличие Microsoft Data Access Components 2.6 Runtime. Нам потребуются компоненты версии 2.6 или выше. Дистрибутивный файл этих компонентов mdac_type.exe можно получить по адресу *http://www.microsoft.com/data/download.htm*.

Программу pgAdmin можно получить с сайта Great Bridge по адресу *http://www.greatbridge.org/project/pgadmin/*. Имя файла имеет вид pgadmin-710.zip и изменяется в зависимости от текущей версии.

Если устанавливать программу в ОС Windows 9x или NT (в отличие от Windows Me или 2000), то потребуется Microsoft Windows Installer (instmsia.exe для Windows 9x или instmsiw.exe для Windows NT), который можно найти по адресу *http://msdn.microsoft.com/downloads/* (поищите «Windows Installer Re-distributable»).

Вот последовательность действий при установке pgAdmin:

- Установить OBDC-драйвер psqlodbc для PostgreSQL
- Установить MDAC версии 2.6 или более поздней из файла mdac_typ.exe
- Установить Windows Installer – в случае Windows 9x или NT (instmsia.exe или instmsiw.exe)
- Разархивировать файл pgAdmin.zip и поместить pgAdmin.msi во временный каталог
- Двойным щелчком мыши по файлу pgAdmin.msi начать установку

В результате установки в стартовом меню появится новая программная группа (pgAdmin). Прежде чем начинать всерьез использовать pgAdmin, необходимо убедиться в том, что можно создавать объекты в той базе, с которой мы собираемся работать. Это необходимо, потому что pgAdmin хранит собственные объекты в базе данных.

Прежде всего, следует удостовериться, что в драйвере ODBC или в источнике данных, который предполагается использовать, включен режим разрешения записи. Для этого запустим еще раз апплет ODBC Data Sources.

Выберем источник данных PostgreSQL и в окне настройки параметров драйвера выключим режим «ReadOnly» (рис. 5.6).

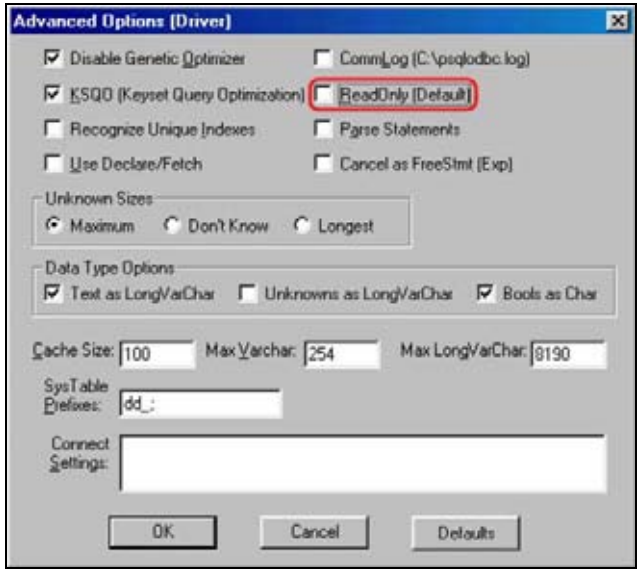


Рис. 5.6. Включаем режим разрешения записи в окне настройки параметров драйвера

Это можно сделать и в окне Advanced Options источника данных (рис. 5.7):

Рис. 5.7. Включаем режим разрешения записи в окне настройки параметров источника данных

Те, кого интересуют внутренние таблицы PostgreSQL, могут включить режим просмотра системных таблиц (флажок Show System Tables).

Теперь мы готовы запустить pgAdmin и начать работу с нашей базой данных. В диалоговом окне регистрации pgAdmin просит ввести источник данных ODBC, а также имя пользователя и пароль (рис. 5.8):

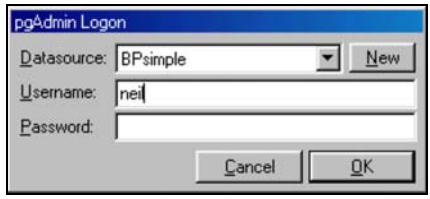


Рис. 5.8. Ввод регистрационной информации

Функции, выполняемые pgAdmin, требуют, чтобы пользователь, под именем которого выполняется регистрация, имел все привилегии в базе данных, другими словами, это должен быть пользователь superuser. При попытке регистрации в качестве пользователя, не имеющего таких полномочий, будет выдано сообщение об ошибке (рис. 5.9):

Рис. 5.9. У регистрирующегося пользователя недостаточно привилегий

Вернемся к рассмотрению пользователей и их привилегий в главе 11. Если PostgreSQL установлен с параметрами по умолчанию, то управление базой данных осуществляется пользователем postgres, и можно зарегистрироваться под этим именем.

Ниже приведен снимок экрана pgAdmin, представляющий список таблиц базы данных bpsimple и свойства поля lname таблицы customer (рис. 5.10):

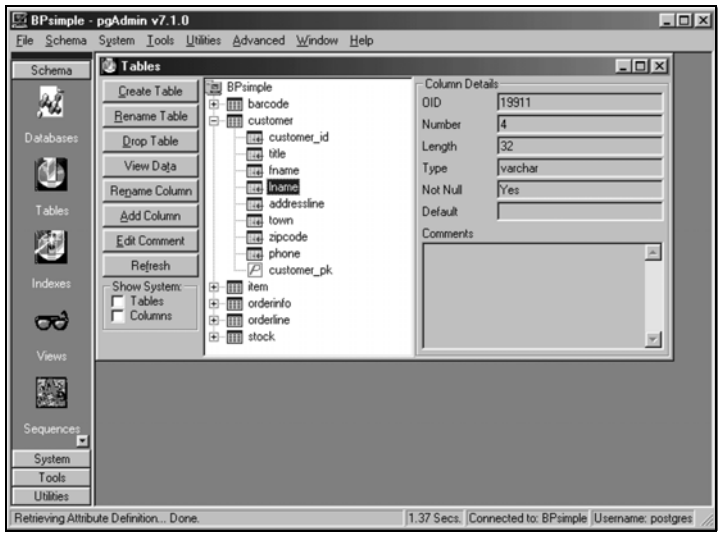


Рис. 5.10. Просмотр базы данных с помощью pgAdmin

Одной из функций pgAdmin, которая может оказаться весьма полезной, является возможность импорта данных. Если у нас имеются какие-либо данные, которые требуется поместить таблицу PostgreSQL, то pgAdmin поможет в этом.

Один из способов импортирования данных требует представления их в виде полей, разделенных запятыми (Comma Separated Variables, CSV). Такие приложения, как Microsoft Excel, могут экспортировать данные в этом формате.

Приведем простой пример. Дополнительные строки уже вставлялись в таблицу numbers базы test при помощи электронной таблицы Excel. Данные были сохранены в формате CSV без заголовков. Это значит, что в первой строке файла нет названий столбцов, а есть только данные. Файл содержит строки:

```
111,Nelson
0,Duck
22,Two Ducks
```

Мастер импорта таблиц pgAdmin вызывается из пункта меню Tools | Import (рис. 5.11):

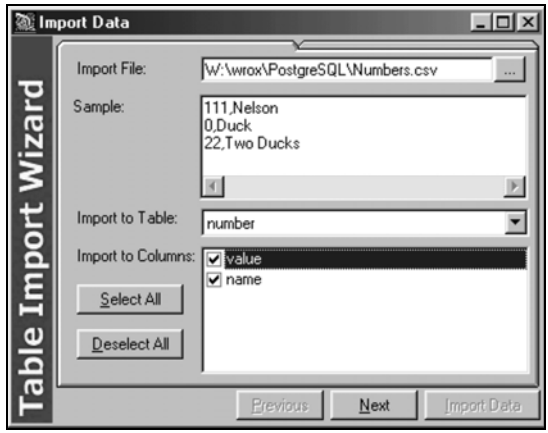


Рис. 5.11. Запущен мастер импортирования данных

Выбираем файл, из которого собираемся импортировать данные, а также таблицы и поля, в которые данные будут импортироваться. Затем необходимо установить некоторые параметры импорта. А именно заключены ли данные в кавычки и каким образом разделяются поля (рис. 5.12).

Осталось нажать кнопку Import Data, и новые строки будут добавлены в таблицу базы данных.

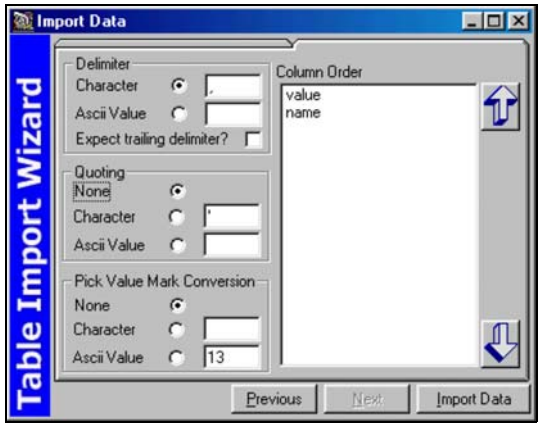


Рис. 5.12. Установка параметров импортирования

Kpsql

В среде Linux графической альтернативой программе `psql` служит `Kpsql` от компании Mutiny Bay Software. Несмотря на то что ее разработка прекращена, программа `Kpsql` остается весьма полезной. Она предназначена для работы в графической оболочке KDE. Ее текущую версию (1.0) можно получить по адресу *http://www.mutinybaysoftware.com*.

По существу, `Kpsql` представляет собой редактор SQL-запросов. Можно создавать, сохранять и редактировать запросы в редакторе с синтаксическим выделением, который окрашивает ключевые слова SQL в различные цвета, это помогает в построении корректных запросов. Полученные запросы можно выполнять и просматривать результаты в отдельном окне с прокруткой (рис. 5.13):

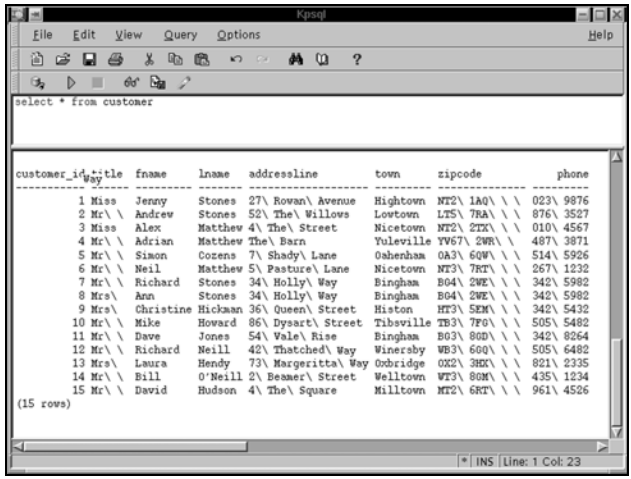


Рис. 5.13. Редактор SQL-запросов Kpsql

Это простой, но эффективный инструмент для тех случаев, когда требуется получить лишь несколько простых отчетов. На снимке экрана показан результат выполнения простого запроса.

Вместе с `Kpsql` поставляются файлы справки, включающие описание работы с редактором запросов и справочник по SQL-командам PostgreSQL.

PgAccess

Официальный дистрибутив PostgreSQL включает в себя утилиту с графическим интерфейсом `PgAccess`, весьма полезную при работе с базами данных PostgreSQL. Программа позволяет:

- Просматривать таблицы, вставлять, удалять и изменять строки
- Создавать новые таблицы и изменять структуры существующих таблиц
- Создавать и выполнять запросы
- Использовать визуальный конструктор для создания запросов и сохранять их в виде представлений
- Проектировать, создавать и исполнять формы ввода данных
- Создавать и модифицировать учетные записи пользователей

Программа `PgAccess` написана на Tcl/Tk и требует, чтобы на клиентской машине был установлен Tcl/Tk версии 8.0 или выше. При выполнении данного требования программа будет работать на любой машине, включая Linux и Microsoft Windows. В этой главе будет рассмотрена работа в Linux, но практически все сказанное относится и к Windows.

Возможно, `PgAccess` не будет установлена по умолчанию. Если ее не удается обнаружить после установки PostgreSQL, то она, возможно, расположена в каталоге `src/bin` дистрибутива с исходными текстами. Также ее можно найти по адресу *http://www.flex.ro/pgaccess*.

Обычно процедура установки заключается в копировании файлов `PgAccess` в какой-либо каталог и редактировании стартового сценария `PgAccess`, в котором определяются значения переменных `PATH_TO_WISH` (путь к `wish` – интерпретатору Tcl/Tk) и `PGACCESS_HOME` (каталог, содержащий файлы `PgAccess`).

Команда запуска `PgAccess` имеет один необязательный параметр – имя базы данных:

```
pgaccess [dbname]
```

Если имя базы данных не указано, `PgAccess` попытается установить соединение с той базой данных, которая была закрыта последней. Ее имя, имя пользователя и другие параметры сохраняются в файле `.pgaccessrc` в вашем домашнем каталоге.

При регистрации в нашей пробной базе данных главное окно PgAccess будет выглядеть приблизительно так (рис. 5.14):

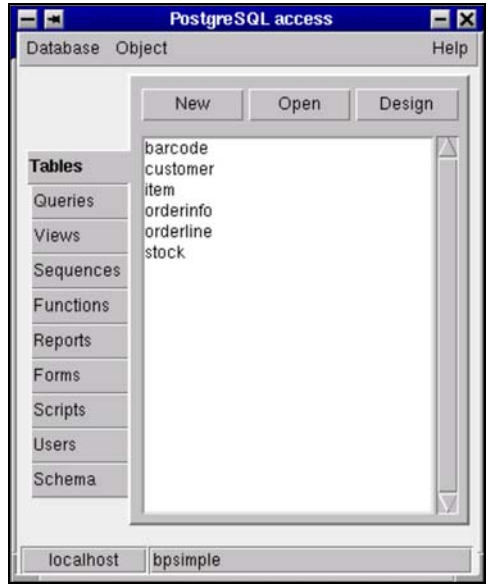


Рис. 5.14. Главное окно PgAccess

Если вы уже работали с программой `pgAdmin`***, то обнаружите в базе данных дополнительные таблицы, в которых*** `pgAdmin` ***хранит свои серверные объекты. Также дополнительные таблицы появляются после применения*** `PgAccess` ***для создания форм.***

Пользовательский интерфейс PgAccess очень прост: в главном окне на вкладках слева представлены функции, а вверху – перечень действий. Для просмотра данных следует выделить интересующую нас таблицу и нажать кнопку Open (рис. 5.15):

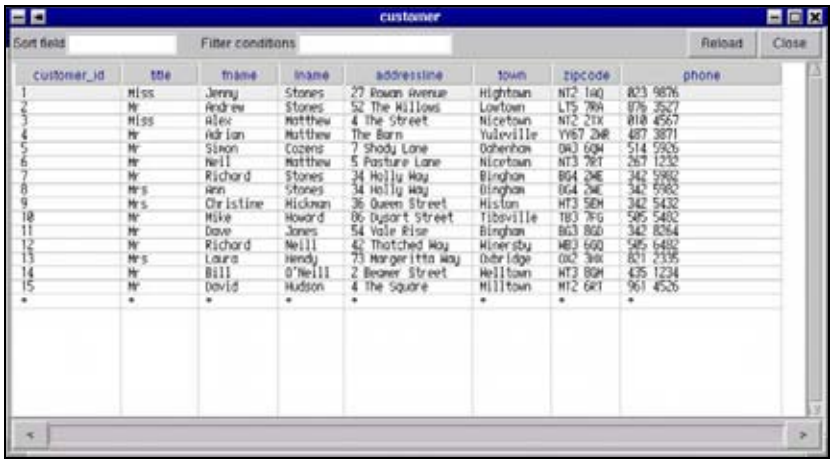


Рис. 5.15. Просмотр таблицы customer в PgAccess

В этом окне можно вносить изменения в данные, удалять и добавлять строки. Для сортировки данных по какому-либо столбцу достаточно ввести его имя в поле Sort Field, а для фильтрации строк следует ввести условие в поле Filter conditions. Здесь показаны записи таблицы `item`, отсортированные по полю `description`, причем только те, в которых значение поля `cost_price` больше двух (рис. 5.16):

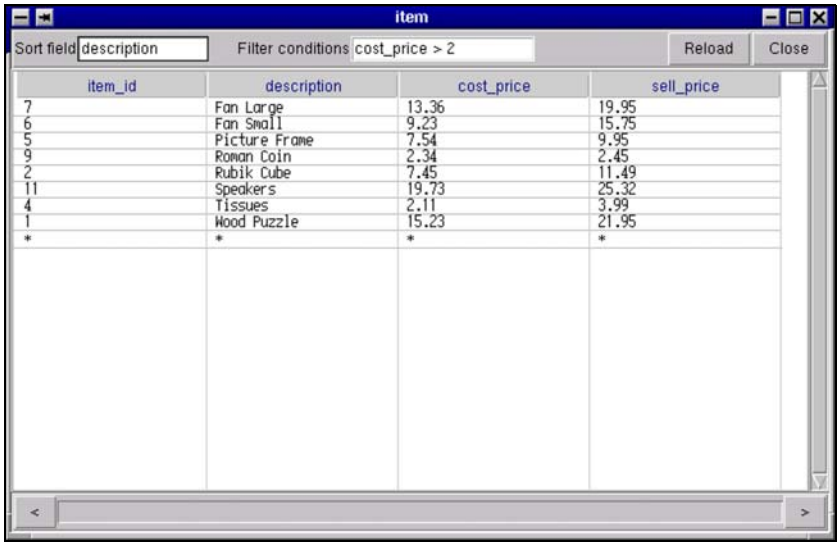


Рис. 5.16. Сортировка с использованием фильтра

Формы и редактор запросов

Возможно, что две наиболее интересные особенности `PgAccess` — это способность создавать и хранить формы и визуальный редактор запросов.

Формы `PgAccess` могут быть созданы в редакторе форм и сохранены в базе данных в виде кода на Tcl/Tk. Сохраненные формы могут быть выполнены с помощью `PgAccess`. Рядом приведен пример формы, взятый из распространяемого вместе с `PgAccess` файла `formdemo.sql`, содержащего демонстрационный код формы. Здесь представлены многие из доступных компонентов пользовательского интерфейса (рис. 5.17):

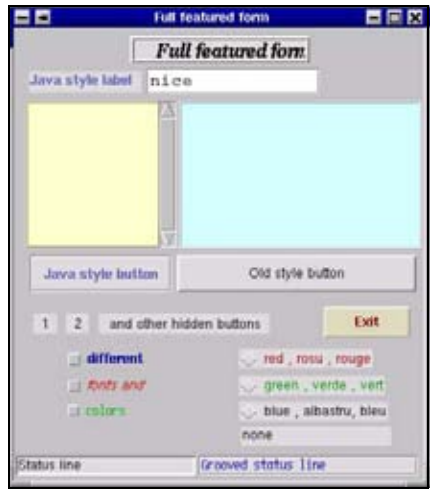


Рис. 5.17. Редактор форм

Добавление кода на Tcl, выполняющегося в момент активации элементов пользовательского интерфейса, таких как кнопки и поля ввода, позволит наиболее полно использовать возможности форм в PgAccess.

Визуальный редактор запросов PgAccess предоставляет возможность графического конструирования SQL-запросов. Несмотря на то что этот редактор поддерживает не все типы запросов, он может быть полезен для предварительного составления сложных запросов. Редактор позволяет перемещать атрибуты из одних таблиц в другие по технологии «Drag and Drop», определяя взаимосвязи таблиц, учитываемые в запросе, и задавать условия на столбцы. Можно также просмотреть получившийся SQL-оператор и выполнить запрос в окне редактирования.

На снимке экрана (рис. 5.18) представлен запрос, показывающий клиентов из определенного города, сделавших заказы:

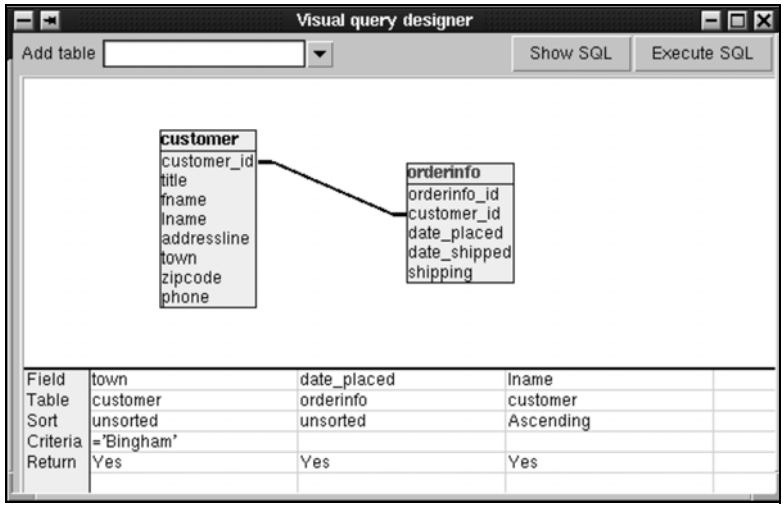


Рис. 5.18. Редактор запросов

Microsoft Access

На первый взгляд это может показаться странным, но Microsoft Access тоже можно использовать совместно с PostgreSQL. Если Access работает с базами данных, то зачем хранить данные в PostgreSQL? И если в PostgreSQL есть такая программа, как PgAccess, то зачем нам еще и Microsoft Access?

Во-первых, при проектировании баз данных необходимо учитывать такие факторы, как объем данных, многопользовательский режим работы, безопасность, устойчивость и надежность. Аргументами в пользу PostgreSQL могут стать его соответствие принятой модели безопасности, серверной платформе и ожидаемому росту объема данных.

Во-вторых, несмотря на то что PostgreSQL, работающий на UNIX- или Linux-сервере, идеально подходит для хранения данных, он может оказаться не самой удобной средой для программ ваших пользователей.

В этом случае пользователям можно предоставить возможность работы с такими программами, как Access, и ей подобными для создания отчетов и форм ввода данных. А так как для PostgreSQL существует ODBC-интерфейс, это не только возможно, но и исключительно просто.

Рассмотрим создание приложения в Access, которое хранит данные на удаленном сервере PostgreSQL и генерирует несложные отчеты по этим данным. Предполагается, что читатели достаточно хорошо знакомы с разработкой баз данных и приложений в Access.

Необходимо установить в Access соединение с PostgreSQL для того, чтобы получить доступ ко всем возможностям Access по созданию простых и удобных приложений, работающих с PostgreSQL.

Связанные таблицы

Для импорта таблиц в собственную базу данных Access предлагает несколько различных путей, один из которых заключается в применении связанных таблиц. Это такие таблицы, которые представлены в Access в виде запросов. Данные не копируются в базу данных, а извлекаются из внешнего источника по мере необходимости. Благодаря такому подходу изменения во внешней базе данных немедленно становятся доступны в Access.

Создадим в Access простую базу данных, позволяющую просматривать и редактировать записи о товарах в нашей учебной базе данных `bpsimple`. У нас есть таблица `item`, в которой хранятся уникальные идентификаторы всех товаров, их описания, а также закупочные и розничные цены.

Создадим в Access новую пустую базу данных (рис. 5.19):

Рис. 5.19. Создание базы данных в Access

На вкладке Tables диалогового окна Database нажмем кнопку New, чтобы создать новую таблицу, затем выберем режим Link Table (рис. 5.20):

Рис. 5.20. Создание новой таблицы

В диалоговом окне Link выберем файлы ODBC Databases (рис. 5.21):

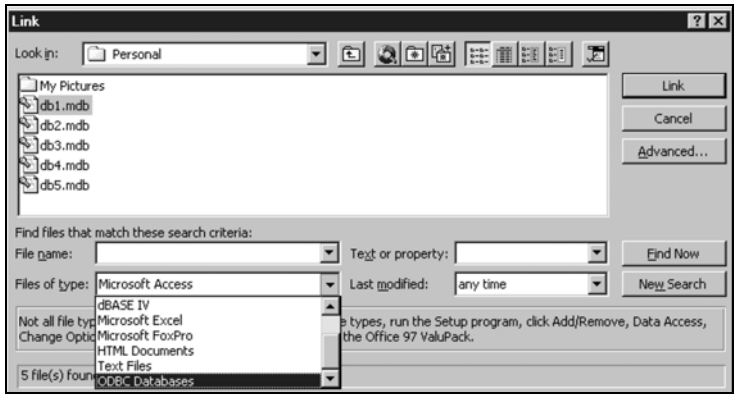


Рис. 5.21. Связывание таблиц

Теперь должно открыться диалоговое окно выбора источника данных ODBC. Перейдите на вкладку Machine DSN и выберите соответствующее соединение с базой данных PostgreSQL (рис. 5.22):

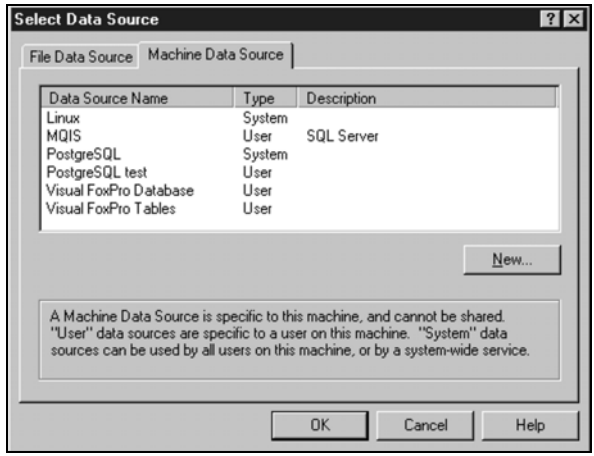


Рис. 5.22. Выбор источника данных ODBC

Теперь следует указать параметры соединения, включая корректное имя пользователя и пароль, которые позволят соединиться с базой данных (рис. 5.23):

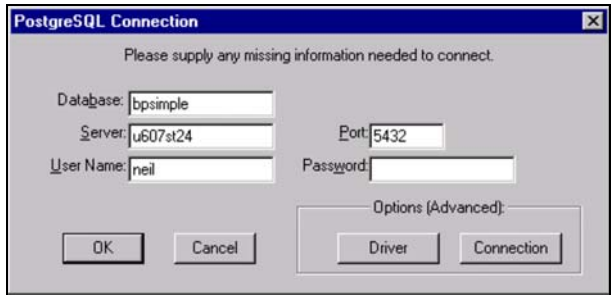


Рис. 5.23. Установка параметров соединения

После установления соединения вам будет предложен список доступных таблиц удаленной базы данных. Выберите `item`, чтобы связать таблицу `item` с базой данных Access (рис. 5.24):

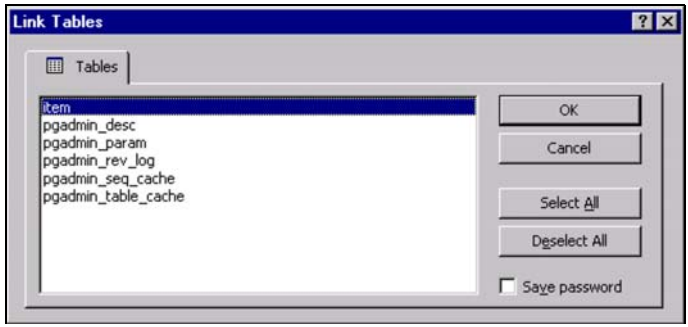


Рис. 5.24. Выбор таблицы для связи с базой данных Access

Прежде чем Access окончательно свяжет таблицу, необходимо указать, какие поля следует использовать для однозначной идентификации записей. Другими словами, из каких полей состоит первичный ключ? В этой таблице первичным ключом является поле `item_id`, поэтому выбираем его. Для составных ключей можно выбрать несколько полей (рис. 5.25):

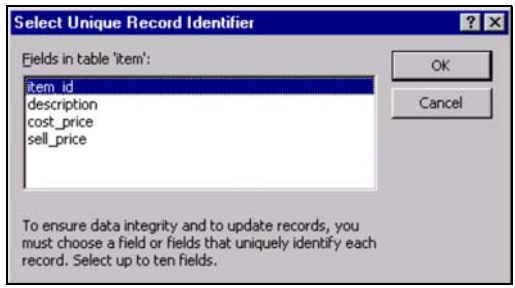


Рис. 5.25. Выбор уникальных идентификаторов записей

Вы видите, что в Access появилась новая таблица, также с именем `item`, с которой можно обращаться так, как если бы данные хранились непосредственно в Access (рис. 5.26):

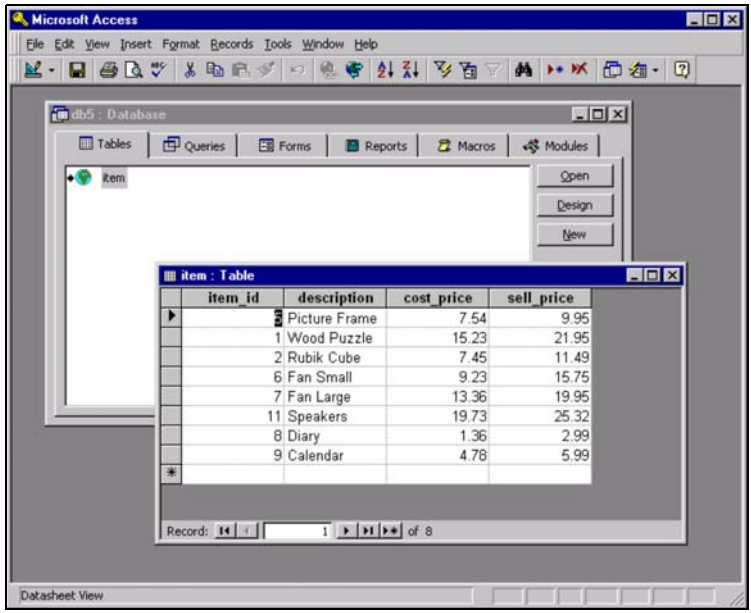


Рис. 5.26. База данных Access

Это все, о чем мы хотели здесь рассказать.

> ***Внешний вид представленных здесь экранов может немного отличаться в зависимости от версий Access и Windows. Если вы увидите в таблице лишний столбец с именем*** `oid`***, можете не обращать на него внимания – это внутренний объект PostgreSQL. Для того чтобы поле*** `object_id` ***не отображалось, отключите параметр OID при настройке источника данных ODBC.***

Ввод данных

Мы можем использовать Access как для просмотра данных в таблицах PostgreSQL, так и для добавления новых строк.

> ***Access может выдавать ошибки при преобразовании некоторых типов данных PostgreSQL, например таких, как*** `NUMERIC(7,2)`***, в собственный формат чисел с плавающей точкой*** `float8`***. Это не позволит выполнить изменение и удаление строк. Если планируется применение Access для работы с данными PostgreSQL, то придется ограничиться простыми типами данных.***

Ниже представлен снимок экрана разработанной в Access формы ввода данных, предназначенной для добавления строк в таблицу `item`. Можно обратиться к возможностям программирования в Access для создания приложений, реализующих более сложные варианты ввода дан-

ных, выполняющие проверку вводимых значений или предотвращающие изменение существующих записей (рис. 5.27):

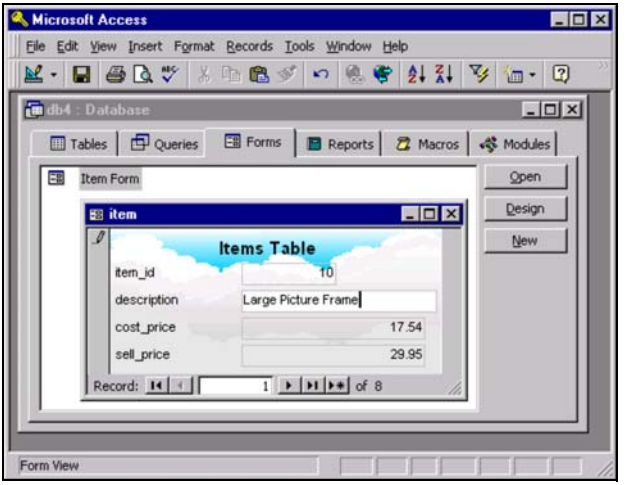


Рис. 5.27. Ввод данных в Access

Отчеты

Отчеты также не вызывают затруднений. Access позволяет создавать отчеты на основе данных, хранимых в таблицах PostgreSQL, точно так же, как и на основе любой таблицы Access. Можно включать в отчет порожденные столбцы для поиска ответов на вопросы о хранимых в таблицах данных. В качестве примера представим отчет, показывающий наценку (разность между `sell_price` и `cost_price`) на товары в таблице `item` (рис. 5.28):

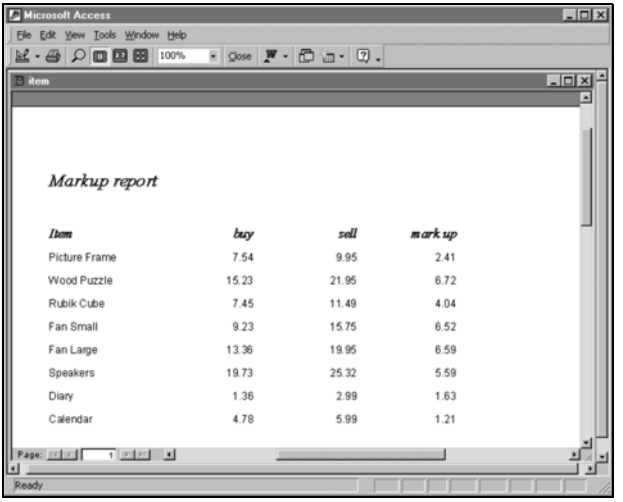


Рис. 5.28. Отчет в Access

Совместное применение Microsoft Access и PostgreSQL расширяет возможности по разработке приложений для баз данных. Масштабируемость и надежность PostgreSQL в сочетании с распространенностью и простотой использования Microsoft Access могут оказаться именно тем, что вам нужно.

Microsoft Excel

Подобно тому как использовался Microsoft Access, можно применять и Microsoft Excel для расширения функциональности PostgreSQL. В основе лежит та же идея, что и в Access: помещаем в электронную таблицу данные, взятые из удаленного источника (или, точнее, связанные с ним). В случае изменения данных можно обновить изображение, и электронная таблица отобразит новые значения.

Связав электронную таблицу с данными PostgreSQL, можно использовать такие возможности Excel, как графическое представление данных в виде диаграмм.

Расширим наш пример с таблицей `product` из Access так, чтобы величина наценки (markup) на товары из таблицы `item` представлялась в виде диаграммы.

Так же как и в Access, необходимо сообщить Excel, что некоторая часть электронной таблицы должна быть связана таблицей внешней базы данных. Создав пустую электронную таблицу, выберем в меню команду формирования запроса к внешней базе данных (рис. 5.29):

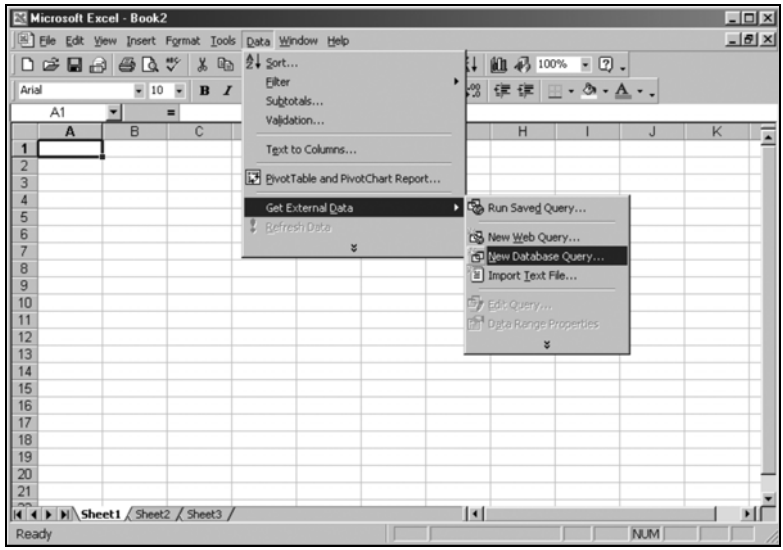


Рис. 5.29. Начинаем формировать запрос

Появится уже знакомый нам диалог выбора источника данных ODBC (рис. 5.30):

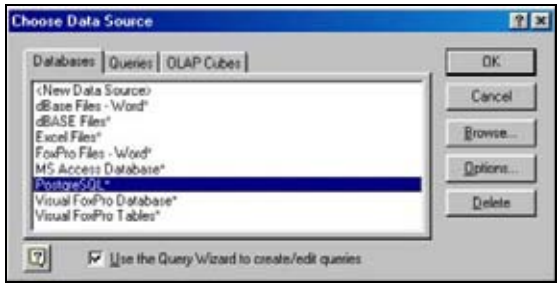


Рис. 5.30. Выбор источника данных

Установив соединение с базой данных, можно выбрать, какую таблицу будем использовать и какие столбцы отображать. Выберем идентификатор товара, его описание и обе цены из таблицы `item` (рис. 5.31):

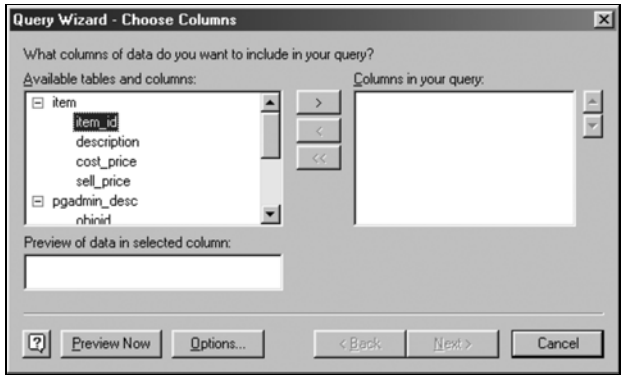


Рис. 5.31. Выбор таблиц и столбцов

Для того чтобы ограничить количество строк, отображаемых в электронной таблице, в следующем диалоговом окне можно задать критерий выбора. Выведем товары стоимостью более двух долларов (рис. 5.32):

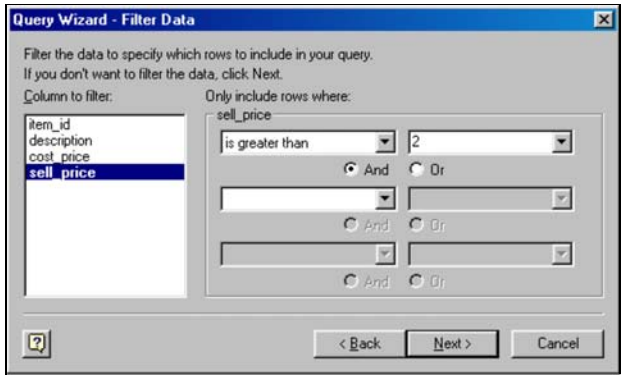


Рис. 5.32. Фильтруем данные

Наконец, можно задать сортировку данных по определенному столбцу или группе столбцов в любом порядке (рис. 5.33):

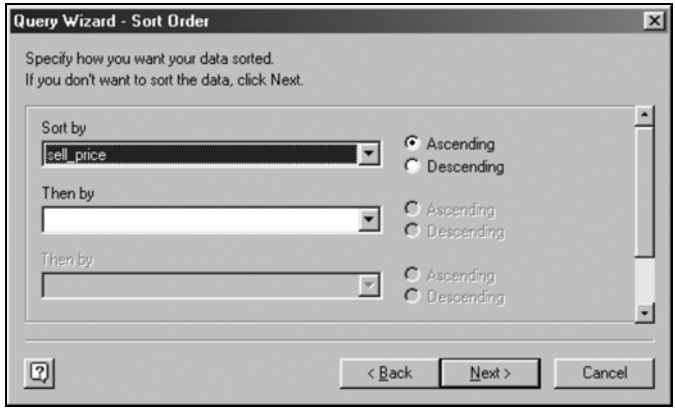


Рис. 5.33. Задаем порядок сортировки

Прежде чем передавать данные в Excel, можно определить, в какой части электронной таблицы будут располагаться данные. Пожалуй, хорошей идеей будет вывод данных из PostgreSQL на отдельный лист. Это нужно для того, чтобы по мере роста базы данных у нас оставалось достаточно места для новых строк. При перезапросе данных в электронной таблице потребуется пространство для новых строк.

В этом примере просто позволяем данным занять левую верхнюю часть листа (рис. 5.34):

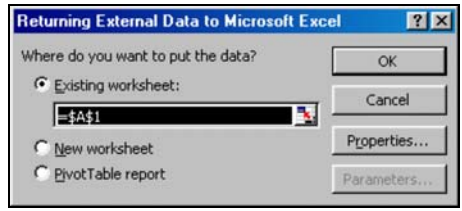
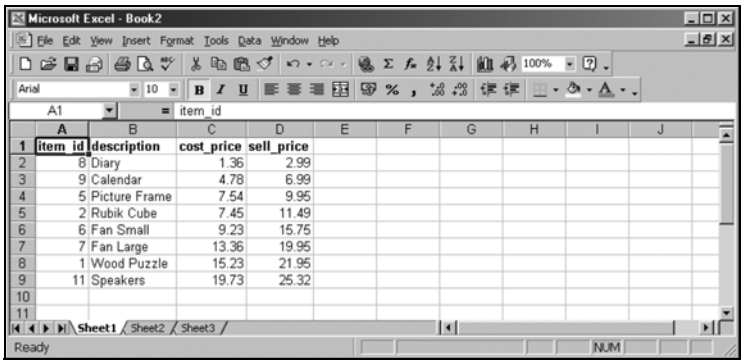


Рис. 5.34. Передача данных в Excel

На листе электронной таблицы появились данные (рис. 5.35):



	A	B	C	D
1	item_id	description	cost_price	sell_price
2	8	Diary	1.36	2.99
3	9	Calendar	4.78	6.99
4	5	Picture Frame	7.54	9.95
5	2	Rubik Cube	7.45	11.49
6	6	Fan Small	9.23	15.75
7	7	Fan Large	13.36	19.95
8	1	Wood Puzzle	15.23	21.95
9	11	Speakers	19.73	25.32
10				
11				

Рис. 5.35. Данные представлены в таблице Excel

При желании в электронной таблице можно произвести вычисления над данными. Например, можно рассчитать наценку на все товары, задав в дополнительном столбце соответствующую формулу.

Изменения в базе данных не вызывают автоматического обновления строк в Excel. Для того чтобы убедиться в актуальности отображаемых данных, необходимо выполнить обновление, выбрав в меню команду Data | Refresh Data (рис. 5.36):

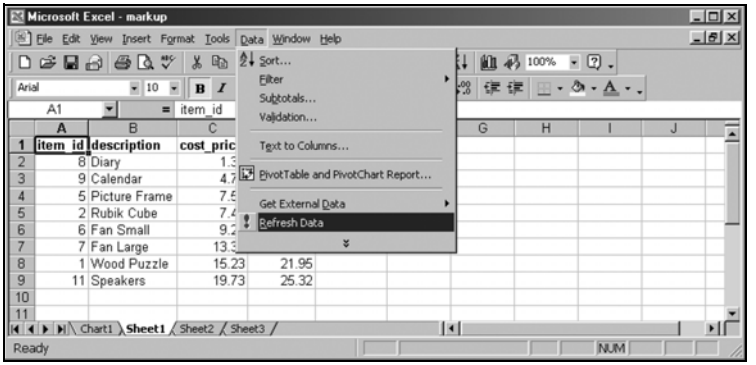


Рис. 5.36. Обновление данных

Теперь в электронной таблице имеются данные, и для их обработки можно воспользоваться средствами Excel. В этом примере построена диаграмма, отображающая величину наценки для каждого из товаров. Для этой цели применен мастер диаграмм Excel, а в качестве исходных данных взяты значения, полученные из PostgreSQL (рис. 5.37):

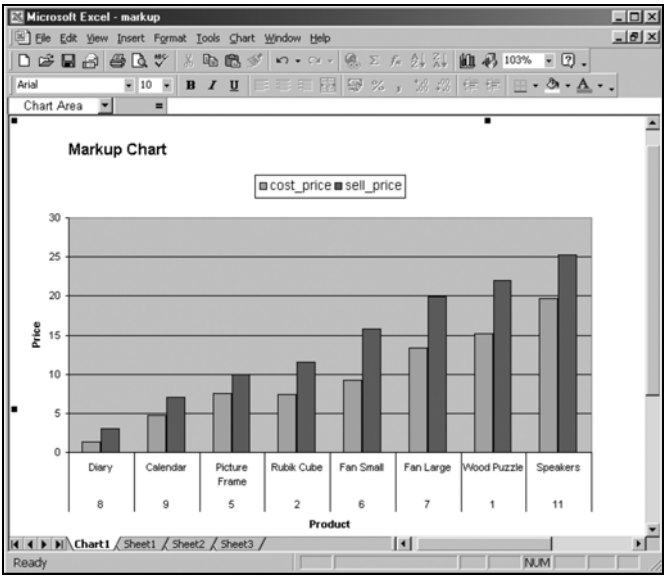


Рис. 5.37. Построение диаграммы

Если после изменения данных в PostgreSQL выполнить обновление в электронной таблице, то диаграмма будет автоматически перерисована.

В некоторых версиях Excel могут возникнуть проблемы с отображением данных при использовании Microsoft Query для соединения с PostgreSQL. Приведенные выше примеры были выполнены в Excel 2000.

Ресурсы

Поиск инструментов для работы с PostgreSQL лучше всего начать с сайта Great Bridge: *http://www.greatbridge.org*.

Программа Zeos database explorer, доступная по адресу *http://www.zeos.dn.ua/eng/*, предназначена для решения общих задач при работе с базами данных и поддерживает PostgreSQL.

Монитор сессий для PostgreSQL под названием `pgmonitor` находится в состоянии разработки и может быть найден по адресу *http://www.greatbridge.org/project/pgmonitor/projdisplay.php*. Это еще одна программа на Tcl/Tk, которая позволяет наблюдать за работой базы данных. Она должна выполняться на сервере базы данных, но позволяет отображать данные на клиентской машине, если на ней установлена X Window System для Unix или Unix-подобной операционной системы.

Резюме

В этой главе рассмотрены некоторые из доступных инструментов для эффективной работы с PostgreSQL. В стандартный дистрибутив входит утилита командной строки `psql`, позволяющая выполнять большинство необходимых операций по созданию и поддержке данных.

При наличии установленного Tcl/Tk для модификации данных можно использовать утилиту с графическим интерфейсом `PgAccess`.

Для целей администрирования с машины под управлением Microsoft Windows можно применять многофункциональную программу `pgAdmin`.

Для создания форм и отчетов, работающих с данными, хранимыми в базе данных PostgreSQL, можно использовать программы, входящие в Microsoft Office, включая Microsoft Excel и Access. Такой подход позволяет объединить надежность и масштабируемость PostgreSQL на платформе UNIX или Linux с простотой применения знакомых инструментов.

6

Работа с данными

В предыдущих главах рассказывалось о том, каким мощным инструментом для организации и поиска данных являются реляционные базы данных, в частности PostgreSQL. Были рассмотрены такие графические средства, как `psql`, `pgAdmin`, `Kpsql` и `PgAccess`, посредством которых можно осуществлять администрирование PostgreSQL. Говорилось и о том, как применять Microsoft Access с PostgreSQL, как расширить функциональность при помощи Microsoft Excel. Конечно, пока в базе данных не содержится никакой информации, все эти сведения не слишком полезны. Вот почему в этой главе мы займемся занесением данных в таблицы PostgreSQL (`INSERT`), обновлением уже имеющихся данных (`UPDATE`) и удалением данных из базы (`DELETE`).

Нам предстоит осилить:

- Добавление данных в базу
- Стандартный оператор `INSERT`
- Вставку данных в столбцы типа `SERIAL`
- Вставку значений `NULL`
- Команду `\copy`
- Загрузку данных непосредственно из другого приложения
- Обновление информации в базе данных
- Удаление данных из базы

Добавление данных в базу

На фоне сложностей оператора SELECT, описанных в главе 4, вставка данных в таблицы PostgreSQL выглядит удивительно просто. Данные добавляются при помощи оператора INSERT. Каждый раз данные могут вставляться только в какую-то одну таблицу и обычно только в одну строку.

Стандартный оператор INSERT

Стандартный SQL-оператор INSERT имеет очень простой синтаксис:

```
INSERT INTO tablename VALUES (list of column values);
```

В списке значений столбцов (list of column values) поля разделены запятыми и указываются в том же порядке, в котором столбцы находятся в таблице.

Простота такого синтаксиса делает его весьма привлекательным, но он также может быть и опасен. Советуем избегать такой конструкции и использовать более надежную, приведенную далее (в ней указываются как значения, так и названия столбцов). Первая синтаксическая структура представлена потому, что она довольно часто применяется, однако мы не рекомендуем пользоваться ею.

Попробуйте сами. Команда \d

Добавим в таблицу customer несколько новых строк. Сначала необходимо выяснить правильный порядок столбцов. Если известно, при помощи какого оператора SQL создана таблица, то все просто, порядок будет таким же, в каком столбцы перечислены в команде CREATE TABLE. Если же (что, к сожалению, бывает очень часто) мы не располагаем таким оператором, то можно обратиться к утилите командной строки psql и с ее помощью получить описание таблицы, используя команду \d. Предположим, требуется посмотреть на структуру таблицы customer в нашей базе данных (об этом рассказано в главе 3). Для того чтобы вывести описание, следует выполнить команду \d. Давайте сделаем это:

```
bpsimple=# \d customer
                                        Table "customer"
  Attribute  |         Type          |                            Modifier
-------------+-----------------------+-------------------------------------
 customer_id | integer               | not null default nextval('"
customer_customer_id_seq"'::text)
 title       | character(4)          |
 fname       | character varying(32) |
 lname       | character varying(32) | not null
 addressline | character varying(64) |
 town        | character varying(32) |
```

```
 zipcode     | character(10)         | not null
 phone       | character varying(16) |
Index: customer_pk
bpsimple=#
```

Отображение выглядит несколько запутанным из-за того, что строки не помещаются на странице и переносятся, но порядок столбцов в таблице `customer` очевиден. Обратите внимание, что столбец `customer_id` описан не совсем так, как было указано в операторе `CREATE TABLE`. Дело в том, что в PostgreSQL существует специальный способ реализации `SERIAL`, который в данном случае и указан для столбца `customer_id`. Пока что запомните просто, что это целое поле. О реализации в PostgreSQL столбцов типа `SERIAL` будет рассказано в главе 8 «Язык определения данных».

Для ввода символьных данных необходимо заключить их в одинарные кавычки (`'`). Если одинарная кавычка находится внутри символьной строки, то для ввода необходимо предварить ее символом обратной косой черты (\). Числа можно вводить «как есть» (если только не требуется в точности сохранить их форму записи). Что касается неизвестных, просто пишем `NULL` (или, в более сложной форме оператора `INSERT`, которая будет описана далее, – просто не предоставляем данных для этого столбца).

Теперь порядок следования столбцов известен, и можно записать оператор `INSERT`:

```
INSERT INTO customer VALUES(16, 'Mr', 'Gavin', 'Smyth', '23 Harlestone',
'Milltown', 'MT7 7HI', '746 3725');
```

Увидим:

```
bpsimple=# INSERT INTO customer VALUES(16, 'Mr', 'Gavin', 'Smyth', '23
bpsimple-# Harlestone', 'Milltown', 'MT7 7HI', '746 3725');
INSERT 18944 1
bpsimple=#
```

Число, непосредственно следующее за `INSERT`, почти наверняка будет другим в вашем случае. Главное, что PostgreSQL вставила новые данные. Первое число в действительности представляет собой внутренний идентификационный номер PostgreSQL, называемый **OID**; обычно он невидим.

> ***Идентификатор объекта (Object IDentification number, OID) – это уникальный, обычно невидимый номер, присваиваемый в PostgreSQL каждой строке. При инициализации базы данных создается счетчик, предназначенный для уникальной нумерации строк. В данном случае была выполнена команда `INSERT`, новой строке был присвоен идентификационный номер `18944`. Число `1` показывает количество добавленных строк. В стандартный SQL номер OID не входит, к тому же нумерация внутри таблицы не является последовательной, так что примем к сведению существование такого идентификатора, но использовать его в приложениях не будем (никогда).***

Проверим, корректно ли вставлены данные (это легко сделать, используя оператор SELECT для извлечения данных):

```
bpsimple=# SELECT * FROM customer WHERE customer_id > 15;
customer_id | title | fname | lname | addressline |  town  | zipcode | phone
------------+-------+-------+-------+-------------+--------+---------+-------
         16 | Mr    | Gavin | Smyth |23 Harlestone|Milltown| MT7 7HI |746 3725
(1 row)

bpsimple=#
```

Из-за ограничений, наложенных на ширину страницы, строки отображения опять-таки переносятся, но видно, что данные добавились правильно.

Предположим, что требуется ввести еще одну строку для клиента по фамилии O'Rourke. Что делать с присутствующей в фамилии одинарной кавычкой?

Используем символ обратной косой черты:

```
INSERT INTO customer VALUES(17, 'Mr', 'Shaun', 'O\'Rourke', '32 Sheepy Lane',
'Milltown', 'MT9 8NQ', '746 3956');
```

Проверим, добавились ли данные:

```
bpsimple=# SELECT * FROM customer WHERE customer_id > 15;
customer_id|title| fname | lname  | addressline  |  town  | zipcode |phone
-----------+-----+-------+--------+--------------+--------+---------+------
        16 | Mr  | Gavin |Smyth   |23 Harlestone |Milltown| MT7 7HI |746 3725
        17 | Mr  | Shaun |O'Rourke|32 Sheepy Lane|Milltown| MT9 8NQ |746 3956
(2 rows)

bpsimple=#
```

Как это работает

Для вставки дополнительной информации в таблицу customer применялся оператор INSERT, при этом значения столбцов указывались в том же порядке, в котором столбцы создавались в таблице. Для того чтобы ввести число, просто напишите его, чтобы добавить строку – заключите ее в одинарные кавычки. Чтобы вставить в строку одинарную кавычку, введите перед ней символ обратной косой черты (\). Если же понадобится добавить символ обратной косой черты, введите два таких символа – \\.

Попробуйте сами. Оператор INSERT

Пусть требуется ввести строку, содержащую не совсем обычный адрес, например: Midtown Street A\33. Как поступить с одиночным символом

обратной косой черты, содержащимся в адресе? Чтобы избежать интерпретации символа обратной косой черты как специального, укажем вторую обратную косую черту:

```
INSERT INTO customer VALUES(18, 'Mr', 'Jeff', 'Baggott', 'Midtown Street
A\\33', 'Milltown', 'MT9 8NQ', '746 3956');
```

Вот как это выглядит:

```
bpsimple=# SELECT * FROM customer WHERE addressline='Midtown Street A\\33';
customer_id|title|fname| lname |     addressline    |  town  |zipcode|  phone
-----------+-----+-----+-------+--------------------+--------+-------+------
        18 |Mr   |Jeff |Baggott|Midtown Street A\33|Milltown|MT9 8NQ|746 3956
(1 row)

bpsimple=#
```

Надежный вариант оператора INSERT

В случае применения операторов `INSERT`, подобных приведенным ранее, не всегда бывает удобно указывать каждый отдельный столбец и быть вынужденным вводить данные именно в том порядке, в котором следуют столбцы в таблице. Возникает элемент риска, ведь случайно можно ввести данные в неправильном порядке, и в результате в базу данных попадет некорректная информация. Так, представьте себе, что в предыдущем примере были случайно перепутаны значения для столбцов `fname` и `lname`. Данные были бы добавлены успешно, поскольку оба столбца принадлежат к текстовому типу, и PostgreSQL не смогла бы обнаружить ошибку. Если затем запросить список фамилий клиентов, в нем появится не `Smyth` (как хотелось бы), а `Gavin`. Одной из важнейших проблем баз данных является некорректный ввод или (как в данном случае) явно ошибочные данные, поэтому следует предпринимать все возможные меры предосторожности, чтобы гарантировать, что в базу попадут только корректные данные. В нашей учебной базе данных из нескольких десятков строк легко обнаружить простые ошибки, но в базах данных, содержащих информацию о десятках тысяч клиентов, находить ошибки (в частности, касающиеся данных, содержащих необычные названия) будет очень сложно.

К счастью, существует несколько модифицированный вариант оператора `INSERT`, с которым проще работать, к тому же он гораздо надежнее:

```
INSERT INTO tablename(list of column names) VALUES (list of column values
corresponding to the column names);
```

В данной разновидности оператора `INSERT` необходимо указывать названия столбцов и значения для них в одном и том же порядке, который может отличаться от порядка, использованного при создании таблицы. В этом случае не надо знать, в каком порядке столбцы были определены в базе данных. В списке практически «друг против друга»

указываются названия столбцов и те данные, которые в эти столбцы нужно вставить. Это удобно и недвусмысленно.

Попробуйте сами. Вставка значений в столбцы

Введем в базу данных еще одну строку, при этом явно зададим названия столбцов:

```
INSERT INTO customer(customer_id, title, fname, lname, addressline, town,
zipcode, phone) VALUES(19, 'Mrs', 'Sarah', 'Harvey', '84 Willow Way',
'Lincoln', 'LC3 7RD', '527 3739');
```

Можно использовать для ввода несколько строк, это упростит чтение и проверку соответствия порядка следования названий столбцов и значений для них.

Выполним этот оператор, набрав его (для удобочитаемости) в несколько строк:

```
bpsimple=# INSERT INTO
bpsimple-# customer(customer_id, title, lname, fname, addressline, town,
bpsimple-# zipcode, phone)
bpsimple-# VALUES(19, 'Mrs', 'Harvey', 'Sarah',  '84 Willow Way', 'Lincoln',
bpsimple-# 'LC3 7RD', '527 3739');
INSERT 22592 1
bpsimple=#
```

Обратите внимание на то, насколько в данном случае легче сравнивать названия полей со вставляемыми в них значениями. Мы сознательно поменяли местами столбцы `fname` и `lname`, чтобы показать, что это возможно. Порядок расположения столбцов может быть любым. Единственное, что важно, – величины должны следовать в том же порядке, что и столбцы.

Заметьте, что приглашение `psql` на ввод команд для последующих строк имеет другой вид и остается таким до тех пор, пока не вводится точка с запятой (указывающая конец команды).

> ***Настоятельно рекомендуем применять только данную форму оператора*** `INSERT` ***– с названиями столбцов, потому что явное указание имен делает работу гораздо более надежной.***

Вставка данных в столбцы типа SERIAL

Пришло время покаяться в небольшом грехе, совершенном в отношении столбца `customer_id`. Ранее нам не было известно, как вводить данные лишь в некоторые столбцы таблицы, оставляя другие нетронутыми. Теперь же, познакомившись со второй формой оператора `INSERT`, использующей названия столбцов, вы можете это сделать (и понять, как важна такая возможность для вставки данных в таблицы, содержащие столбцы типа `SERIAL`).

В главе 2 (где впервые упоминался этот достаточно специфический тип) говорилось, что на самом деле SERIAL – это целое число, которое автоматически увеличивается, обеспечивая создание уникальных номеров unique_id для каждой строки. Вставляя данные в строки (ранее в этой главе), мы предоставляли значение и для столбца customer_id, поля данных типа SERIAL.

Посмотрим на данные, содержащиеся в таблице customer в настоящий момент:

```
bpsimple=# SELECT customer_id, fname, lname, addressline FROM customer;
 customer_id |   fname   |  lname   |     addressline
-------------+-----------+----------+---------------------
           1 | Jenny     | Stones   | 27 Rowan Avenue
           2 | Andrew    | Stones   | 52 The Willows
           3 | Alex      | Matthew  | 4 The Street
           4 | Adrian    | Matthew  | The Barn
           5 | Simon     | Cozens   | 7 Shady Lane
           6 | Neil      | Matthew  | 5 Pasture Lane
           7 | Richard   | Stones   | 34 Holly Way
           8 | Ann       | Stones   | 34 Holly Way
           9 | Christine | Hickman  | 36 Queen Street
          10 | Mike      | Howard   | 86 Dysart Street
          11 | Dave      | Jones    | 54 Vale Rise
          12 | Richard   | Neill    | 42 Thatched way
          13 | Laura     | Hendy    | 73 Margeritta Way
          14 | Bill      | O'Neill  | 2 Beamer Street
          15 | David     | Hudson   | 4  The Square
          16 | Gavin     | Smyth    | 23 Harlestone
          17 | Shaun     | O'Rourke | 32 Sheepy Lane
          18 | Jeff      | Baggott  | Midtown Street A\33
          19 | Sarah     | Harvey   | 84 Willow Way
(19 rows)

bpsimple=#
```

Казалось бы, все хорошо, но есть небольшая проблема. Принудительно задав значения для столбца customer_id, мы неумышленно оставили в неправильном состоянии внутренний счетчик SERIAL, существующий в PostgreSQL.

Попробуем ввести еще одну строку, на этот раз предоставим типу SERIAL возможность автоматически увеличить и ввести значение для столбца customer_id:

```
bpsimple=# INSERT INTO customer(title, fname, lname, addressline, town,
bpsimple-# zipcode, phone)
bpsimple-# VALUES('Mr', 'Steve', 'Clarke',  '14 Satview way', 'Lincoln',
bpsimple-# 'LC4 3ED', '527 7254');
ERROR:  Cannot insert a duplicate key into unique index customer_pk

bpsimple=#
```

Очевидно, что здесь что-то не так, ведь никаких повторяющихся значений мы не вводили. Ответ кроется в действиях, совершенных ранее, – задавая значения для столбца customer_id, мы таким образом аннулировали автоматическое назначение PostgreSQL идентификаторов для столбца типа SERIAL, а это привело к тому, что система автоматического распределения идентификаторов перестала соответствовать реальным данным таблицы.

Не предоставляйте значений для столбцов типа `SERIAL` ***при вставке данных.***

Как же выбраться из этой неприятной ситуации? Надо протянуть PostgreSQL руку помощи, синхронизировать ее внутреннюю последовательность номеров с реальными данными.

Доступ к значениям последовательности

При создании таблицы customer столбец customer_id был определен как принадлежащий к типу SERIAL. Вы могли заметить, что при этом PostgreSQL выдала информационные сообщения, в которых говорилось, что создается последовательность customer_customer_id_seq. Кроме того, если запросить у PostgreSQL описание таблицы, указав ключ \d, будет видно, что столбец определен особенным образом:

```
customer_id integer not null default nextval('customer_customer_id_seq'::text)
```

PostgreSQL создала специальный счетчик для столбца – **последовательность**, которая может использоваться для генерирования уникальных идентификаторов. Обратите внимание, что последовательность (sequence) всегда называется следующим образом: <tablename>_<column name>_seq. PostgreSQL автоматически указывает, что по умолчанию значения столбца должны быть результатом выполнения функции nextval('customer_customer_id_seq'). Если в операторе INSERT не вводятся данные для такого столбца, то PostgreSQL автоматически выполняет вышеуказанную функцию. Если же предоставить данные для столбца, то тем самым будет сорвано исполнение автоматического алгоритма (если данные предоставлены, функция не будет вызвана). К счастью, нет необходимости удалять из таблицы все данные и начинать заново, в PostgreSQL есть возможность прямого управления значениями последовательности.

Значение последовательности можно узнать с помощью функции currval:

```
currval('sequence name');
```

PostgreSQL выдаст текущее значение последовательности.

Например:

```
bpsimple=# SELECT currval('customer_customer_id_seq');
 currval
-------------
```

```
      17
(1 row)

bpsimple=#
```

Как видите, PostgreSQL считает, что текущий номер последней строки таблицы равен 17, но на самом деле последняя строка имеет номер 19. При попытке добавить данные в таблицу customer и оставить столбец customer_id на усмотрение PostgreSQL СУБД пытается предоставить значение для столбца, вызывая функцию nextval():

```
nextval('sequence number');
```

Эта функция сначала увеличивает переданное ей значение последовательности, затем возвращает результат. Посмотрим, что происходит:

```
bpsimple=# SELECT nextval('customer_customer_id_seq');
 nextval
---------
      18
(1 row)

bpsimple=#
```

Конечно, можно было бы получить правильное значение для последовательности, несколько раз выполнив nextval(), но такой способ не очень полезен в случае, если значение очень большое или, наоборот, очень маленькое. Вместо этого используем функцию setval():

```
setval('sequence number', new value);
```

Сначала необходимо определить, каким должно быть значение последовательности. Для этого выбираем максимальное значение столбца, присутствующее в базе данных. Существует специальная функция MAX(column name) (поговорим о ней в главе 7), работать с которой чрезвычайно просто – она выдает максимальное числовое значение, сохраненное в столбце:

```
bpsimple=# SELECT MAX(customer_id) FROM customer;
max
-----
  19

bpsimple=#
```

Итак, PostgreSQL в ответ выводит самое большое число, которое будет обнаружено в столбце customer_id таблицы customer. Теперь можно задать последовательность с помощью функции setval(sequence, value), которая позволяет установить последовательность в любое выбранное значение. Самым большим текущим значением таблицы является 19, а номер последовательности всегда увеличивается, прежде чем его значение начинает использоваться, так что обычно последовательность

должна иметь номер, равный максимальному текущему значению таблицы:

```
bpsimple=# SELECT setval('customer_customer_id_seq', 19);
 setval
--------
     19
(1 row)

bpsimple=#
```

Теперь значение последовательности корректно, можно вводить наши данные, предоставив возможность PostgreSQL самостоятельно определять значение для столбца `customer_id` типа `SERIAL`:

```
bpsimple=# INSERT INTO customer(title, fname, lname, addressline, town,
bpsimple-# zipcode, phone) VALUES('Mr', 'Steve', 'Clarke',  '14 Satview
bpsimple-# way', 'Lincoln', 'LC4 3ED', '527 7254');
INSERT 21459 1

bpsimple=#
```

Получилось! Последовательность PostgreSQL опять синхронизирована, и значения типа `SERIAL` по-прежнему будут создаваться правильно.

Проблемы, связанные с тем, что данные не синхронизируются с последовательностью, встречаются достаточно редко. В основном они возникают по следующим причинам:

- Удалена и заново создана таблица, а последовательность не была удалена и создана заново.
- Возникла путаница – вместо того чтобы при вводе данных позволить PostgreSQL подбирать значения для столбцов типа `SERIAL`, вы сами явно указали значение для такого столбца.

Ввод значений NULL

В главе 2 говорилось, что значения `NULL` могут добавляться в таблицу при помощи оператора `INSERT`. Остановимся на этом подробнее.

В первой форме оператора `INSERT` (данные вводятся в столбцы в том порядке, в каком столбцы были определены при создании таблицы) можно просто написать `NULL` вместо значения для столбца. Кавычки не обязательны, т. к. `NULL` – это не символьная строка. Не забывайте, что `NULL` в SQL является специальным неопределенным значением, что совсем не то же самое, что пустая строка.

Представим себе, что в предыдущем примере:

```
INSERT INTO customer VALUES(16, 'Mr', 'Gavin', 'Smyth', '23 Harlestone',
'Milltown', 'MT7 7HI', '746 3725');
```

неизвестно имя клиента. Определение таблицы разрешает наличие NULL в столбце fname, так что абсолютно законно можно вводить данные, не зная имени клиента. Если написать:

```
INSERT INTO customer VALUES(16, 'Mr', '', 'Smyth', '23 Harlestone',
'Milltown', 'MT7 7HI', '746 3725');
```

то получится не то, что планировалось; в качестве имени будет введена пустая строка, тем самым, вероятно, будет подразумеваться, что у данного клиента (Mr. Smyth) нет имени, на самом же деле нужно было использовать NULL, потому что имя нам неизвестно.

Вот как должен выглядеть правильный оператор INSERT для данного случая:

```
INSERT INTO customer VALUES(16, 'Mr', NULL, 'Smyth', '23 Harlestone',
'Milltown', 'MT7 7HI', '746 3725');
```

Обратите внимание на то, что кавычки вокруг NULL отсутствуют. Если использовать кавычки, fname будет установлен в строку NULL, а не в *значение* NULL.

Во второй (более надежной) форме оператора INSERT (в которой явно называются столбцы) вводить значения NULL гораздо легче, просто не указываем ни столбец, ни значение для него:

```
INSERT INTO customer(title, lname, addressline, town, zipcode, phone)
VALUES('Mr', 'Smyth', '23 Harlestone', 'Milltown', 'MT7 7HI', '746
3725');
```

Видите, столбец fname не упомянут в списке, значение для него также не определено. Возможен и другой вариант – включить столбец в список и внести NULL в список значений.

Если попытаться ввести значение NULL в столбец, который определен, как не допускающий использования NULL, ничего не получится. Попробуем, например, добавить клиента без данных для столбца lname (фамилия):

```
bpsimple=# INSERT INTO customer(title, fname, addressline, town, zipcode,
bpsimple-# phone) VALUES('Ms', 'Gill', '27 Chase Avenue', 'Lowtown', 'LT5
bpsimple-# 8TQ', '876 1962');
ERROR:  ExecAppend: Fail to add null value in not null attribute lname

bpsimple=#
```

Не было предоставлено значение для lname, и INSERT был отклонен, поскольку в определении таблицы customer содержится запрет на использование NULL в данном столбце.

Давайте проверим:

```
bpsimple=# \d customer
                                Table "customer"
  Attribute  |    Type     |                      Modifier
-------------+-------------+-------------------------------------
 customer_id | integer     | not null default
nextval('customer_customer_id_seq'::text)
 title       | char(4)     |
 fname       | varchar(32) |
 lname       | varchar(32) | not null
 addressline | varchar(64) |
 town        | varchar(32) |
 zipcode     | char(10)    | not null
 phone       | varchar(16) |
Index: customer_pk

bpsimple=#
```

В главе 8 «Язык определения данных» будет рассказано о том, как определять явные значения по умолчанию, которые будут подставляться в столбцы, если данные вводятся без значения для этих столбцов (указывается значение по умолчанию для столбца).

Команда \copy

Несмотря на то что стандартным способом добавления информации в базу данных в SQL является применение оператора INSERT, это не всегда бывает удобно.

Предположим, что в базу данных требуется добавить большое количество строк, при этом данные уже существуют, скажем, в электронной таблице. До этого момента единственным известным вам способом введения данных в базу было применение команды Export, поэтому, вероятнее всего, электронная таблица была бы экспортирована как файл значений, разделенных запятыми. Затем можно было бы использовать текстовый редактор, такой как EMACS (или какой-то другой, имеющий макросредства), для конвертирования данных в операторы INSERT.

Пусть первоначально данные выглядят так:

```
Miss,Jenny,Stones,27 Rowan Avenue,Hightown,NT2 1AQ,023 9876
Mr,Andrew,Stones,52 The Willows,Lowtown,LT5 7RA,876 3527
Miss,Alex,Matthew,4 The Street,Nicetown,NT2 2TX,010 4567
```

Можно преобразовать их в ряд операторов INSERT, и тогда они примут следующий вид:

```
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Miss','Jenny','Stones','27 Rowan Avenue','Hightown','NT2 1AQ','023
9876');
```

```
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mr', 'Andrew', 'Stones', '52 The Willows', 'Lowtown', 'LT5 7RA', '876
3527');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Miss', 'Alex', 'Matthew', '4 The Street', 'Nicetown', 'NT2 2TX', '010
4567');
```

Сохраним данные в текстовом файле с расширением `.sql`.

Операторы в файле можно выполнить при помощи команды `\i` в `psql`. Так работает файл `pop_customer.sql` (это использовалось в главе 3 для начального заполнения базы данными). Обратите внимание, что генерация уникального значения `customer_id` поручена PostgreSQL.

Надо сказать, что описанный метод не очень удобен. Было бы гораздо лучше, если бы можно было перемещать данные между плоскими файлами и базой данных более простым способом. В PostgreSQL есть две возможности сделать это. При этом обе реализуются командой `copy` (что вносит некоторую путаницу). В PostgreSQL существует команда SQL, называемая `COPY`, она может сохранять и восстанавливать данные в плоские файлы, но применять эту команду может только администратор базы данных, при этом файлы читаются с сервера и записываются на сервер. Больше пользы от универсальной команды `\copy`, которая реализует почти все функции `COPY`, а выполняться может кем угодно, при этом данные читаются и записываются на клиентской машине. Следовательно, SQL-команду `COPY` можно считать избыточной, поэтому посвятим ей следующий абзац и больше не будем о ней упоминать.

У команды `COPY` есть одно важное преимущество. Она работает значительно быстрее, поскольку выполняется непосредственно в серверном процессе, тогда как команда `\copy` выполняется в клиентском процессе, при этом все данные должны передаваться по сети. В случае ошибок она также чуть более надежна. Однако до тех пор пока речь не идет об очень больших объемах данных, разница не слишком заметна.

Приведем стандартный синтаксис команды `\copy` для импортирования данных:

```
\copy tablename FROM 'filename' [USING DELIMITERS 'a single character to use
as a delimiter'] [WITH NULL AS 'a string that means NULL']
```

Выглядит несколько громоздко, но работать с ней очень просто. Разделы, заключенные в квадратные скобки `[]`, не обязательны, так что используйте их лишь при необходимости. Запомните, что название файла должно быть заключено в одинарные кавычки.

Параметр `USING DELIMITERS 'a single character...'` позволяет указать, каким образом отделяется каждый столбец (какой символ выступает в качестве разделителя) во входном файле. По умолчанию считается, что столбцы входного файла разделены знаками табуляции. В нашем случае будем основываться на предположении о том, что стартовым является файл значений, разделенных запятыми, полученный в результате

экспорта данных из электронной таблицы. На практике часто оказывается, что такой формат (значения, разделенные запятыми) – это не лучший выбор, потому что запятая как символ может встретиться в данных. В частности, адресная информация почти всегда содержит запятые. К сожалению, электронные таблицы редко предлагают разумные альтернативы экспорту файлов значений, разделяемых запятыми, поэтому может случиться так, что придется работать с тем что есть. Если же есть выбор, удобно использовать символ конвейера «|», т. к. он чрезвычайно редко встречается в данных пользователя.

Параметр `WITH NULL AS 'a string...'` позволяет указать строку, которая должна интерпретироваться как неизвестная (`NULL`). По умолчанию за ее значение принимается `\N`. Обратите внимание, что в команде `\copy` строки должны заключаться в кавычки, таким образом PostgreSQL получает информацию о том, что это строка, в реальных же данных наличие кавычек не предполагается. Если в качестве значения `NULL` требуется загрузить в базу данных строку `NOTHING`, то следует задать параметр `WITH NULL AS 'NOTHING'`, тогда данные (если, например, неизвестно имя Mr. Hudson) должны выглядеть следующим образом:

```
15,Mr,NOTHING,Hudson,4 The Square, Milltown,MT2 6RT,961 4526
```

При непосредственном вводе данных следует проявить внимание и осторожность, чтобы обеспечить «чистоту» данных. Необходимо гарантировать, что нет пропущенных столбцов, что перед всеми кавычками, которые должны присутствовать в данных, поставлены символы обратной косой черты, что нет символов в двоичном представлении и т. д. PostgreSQL обнаружит большую часть подобных ошибок за вас и загрузит только корректные данные, но «распутывать» тысячи строк данных, которые уже почти полностью были загружены, – это долгий, ненадежный и малорентабельный труд. Стоит попытаться максимально очистить данные *до того*, как загружать их все сразу с помощью команды `\copy`.

Попробуйте сами. Загрузка данных командой \copy

Предположим, что некоторые дополнительные сведения о клиентах содержатся в файле `cust.txt`, представленном ниже:

```
21,Miss,Emma,Neill,21 Sheepy Lane,Hightown,NT2 1YQ,023 4245
22,Mr,Gavin,Neill,21 Sheepy Lane,Hightown,NT2 1YQ,023 4245
23,Mr,Duncan,Neill,21 Sheepy Lane,Hightown,NT2 1YQ,023 4245
```

Удачно, что в данном случае не надо заботиться о неизвестных (их нет), поэтому необходимо задать только запятую в качестве разделителя столбцов. Для загрузки данных просто выполняем команду:

```
\copy customer from 'cust.txt' using delimiters ','
```

Отметьте отсутствие в команде точки с запятой (;). Дело в том, что это команда непосредственно psql, а не SQL. В ответ psql выдает краткое \., сообщая, что все хорошо.

Если теперь запустить

```
SELECT * FROM customer;
```

то будет видно, что дополнительные строки добавлены.

Все же здесь таится одна небольшая неприятность. Помните, что может произойти рассинхронизация значений последовательности? К сожалению, применение команды \copy для загрузки данных представляет собой один из способов, приводящих к такому результату. Проверим, что произошло со значением последовательности:

```
bpsimple=# SELECT MAX(customer_id) FROM customer;
 max
-----
  23
(1 row)

bpsimple=# SELECT currval('customer_customer_id_seq');
 currval
---------
      20
(1 row)

bpsimple=#
```

Нехорошо! Максимальное значение, сохраненное в customer_id, в настоящий момент равно 22, и следующий ID должен иметь номер 23, последовательность же попытается присвоить в качестве следующего значения номер 21. Не беспокойтесь, это легко исправить:

```
bpsimple=# SELECT setval('customer_customer_id_seq', 23);
 setval
--------
     23
(1 row)

bpsimple=#
```

Как это работает

Для непосредственной загрузки данных, которые были экспортированы из электронной таблицы в формате списка значений, разделенных запятыми, в таблицу customer, была использована команда \copy. Впоследствии пришлось подправить номер последовательности, генерируемый как значение для столбца customer_id типа SERIAL, но объем работ был значительно меньше, чем тот, который пришлось бы реализовать для конвертирования данных из списка значений, разделенных запятыми, в последовательность операторов INSERT.

Загрузка данных напрямую из другого приложения

Если данные находятся в базе данных Microsoft Access, существует и еще более простой способ загрузки данных в PostgreSQL. Можно просто присоединить PostgreSQL к базе данных Access по ODBC и вставить данные в таблицу PostgreSQL.

Часто оказывается, что существующие данные – это не совсем то, что вам нужно, или что надо их немного доработать, прежде чем вставлять в целевую таблицу. Даже если данные имеют правильный формат, часто бывает разумнее не пытаться вставить их непосредственно в базу, а сначала переместить в таблицу загрузки, а затем уже переслать из таблицы загрузки в реальную таблицу. Использование промежуточной таблицы загрузки является общепринятым для вставки информации в базу данных в реальных приложениях, в частности, если есть сомнения в качестве исходных данных. Данные сначала загружаются в базу данных в промежуточную таблицу-хранилище, проверяются, в случае необходимости исправляются, а затем перемещаются в окончательную таблицу.

Обычно приходится писать специальное приложение или хранимую процедуру (см. главу 10 «Хранимые процедуры и триггеры») для проверки и корректирования информации. Когда данные готовы к загрузке в конечную таблицу, используется удобная для данной ситуации разновидность команды `INSERT`, позволяющая перемещать данные из таблицы в таблицу, при этом в одной команде пересылается сразу несколько строк. Только в этом, исключительном случае один оператор `INSERT` распространяется на несколько строк. Такой оператор называется `INSERT INTO`.

Для вставки данных из одной таблицы в другую применяется нижеприведенный синтаксис:

```
INSERT INTO tablename(list of column names) SELECT normal select statement
```

Предположим, есть некоторая специальная таблица, `tcust`, содержащая дополнительные сведения о клиентах, которые должны быть загружены в главную таблицу `customer`.

Определим эту промежуточную таблицу следующим образом:

```
create table tcust
(
    title                   char(4)          ,
    fname                   varchar(32)      ,
    lname                   varchar(32)      ,
    addressline             varchar(64)      ,
    town                    varchar(32)      ,
    zipcode                 char(10)         ,
    phone                   varchar(16)
);
```

Обратите внимание, что здесь нет ни первичных ключей, ни какого-либо рода ограничений. Это обычная ситуация при кросс-загрузке данных в промежуточную таблицу. Делается все для того, чтобы получить данные в такую таблицу было как можно проще. Отсутствие ограничений и есть такое упрощение. Отметьте, что представлены все необходимые столбцы, кроме значений последовательности `customer_id`, – этот столбец создаст за нас PostgreSQL при загрузке данных.

Предположим, что какие-то данные были загружены в `tcust` (неважно как: по ODBC, с помощью `\copy` или каким-то еще способом), проверены на достоверность и поправлены, тогда вывод `SELECT` будет выглядеть так:

```
bpsimple=# SELECT * FROM tcust;
 title | fname |  lname   |  addressline   |  town   |  zipcode   |  phone
-------+-------+----------+----------------+---------+------------+----------
 Mr    | Peter | Bradley | 72 Milton Rise | Keynes  | MK41 2HQ   |
 Mr    | Kevin | Carney  | 43 Glen Way    | Lincoln | LI2 7RD    | 786 3454
 Mr    | Brian | Waters  | 21 Troon Rise  | Lincoln | LI7 6GT    | 786 7243
(3 rows)

bpsimple=#
```

Попробуйте сами. Загрузка данных из таблицы в таблицу

Первое, что бросается в глаза – для `Mr. Bradley` не удалось найти телефонный номер. Это может вызвать затруднение, а может и не вызвать. Пока что решим не загружать данную строку, а все остальные строки – загружать. На практике (при решении реальной задачи) может понадобиться загрузить сведения о сотнях новых клиентов, и вполне вероятно, что будет удобно осуществлять загрузку по частям, для групп клиентов, после того как информация о каждой такой группе будет признана достоверной или скорректирована.

Написать первую часть оператора `INSERT` достаточно просто. Выберем полную форму оператора, точно указывая названия столбцов для загрузки. Обычно именно такое решение является самым разумным:

```
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone);
```

Обратите внимание, что столбец `customer_id` не указан в списке. Как вы помните, таким образом мы предоставляем PostgreSQL возможность автоматически создать значения для этого столбца, именно такой способ является наиболее надежным для создания значения типа `SERIAL`.

Теперь надо написать часть с оператором `SELECT`, который сформирует данные для `INSERT`. Помните, решено было пока не вставлять данные о `Mr. Bradley`, поскольку его телефонный номер определен как `NULL`, мы еще попытаемся его найти. При желании можно было бы загрузить и `Mr. Bradley`, поскольку столбец телефонных номеров допускает исполь-

зование значений NULL. Просто применим в данном случае бизнес-правило относительно телефона, которое будет более строгим, чем низкоуровневое правило базы данных для хранения номера телефона. Запишем оператор SELECT следующим образом:

```
SELECT title, fname, lname, addressline, town, zipcode, phone FROM tcust
WHERE phone IS NOT NULL;
```

Конечно же, этот оператор сам по себе вполне законен. Протестируем его:

```
bpsimple=# SELECT title, fname, lname, addressline, town, zipcode, phone
bpsimple-# FROM tcust WHERE phone IS NOT NULL;
 title | fname | lname  |  addressline  |  town   |  zipcode  |  phone
-------+-------+--------+---------------+---------+-----------+----------
 Mr    | Kevin | Carney | 43 Glen Way   | Lincoln | LI2 7RD   | 786 3454
 Mr    | Brian | Waters | 21 Troon Rise | Lincoln | LI7 6GT   | 786 7243
(2 rows)

bpsimple=#
```

Кажется, все правильно. SELECT находит те строки, которые требуется найти, столбцы имеют тот же порядок, что и в операторе INSERT. Соединим два оператора вместе и выполним:

```
bpsimple=# INSERT INTO customer(title, fname, lname, addressline, town,
bpsimple-# zipcode, phone) SELECT title, fname, lname, addressline, town,
bpsimple-# zipcode, phone FROM tcust WHERE phone IS NOT NULL;
INSERT 0 2

bpsimple=#
```

Видите, psql сообщает, что две строки были вставлены. Теперь, соблюдая все меры предосторожности, осуществим выборку этих строк из таблицы customer, исключительно для того, чтобы окончательно убедиться в том, что они были загружены корректно:

```
bpsimple=# SELECT customer_id, fname, lname, addressline FROM customer WHERE
bpsimple-# town = 'Lincoln';
 customer_id | fname | lname  |  addressline
-------------+-------+--------+---------------
          18 | Sarah | Harvey | 84 Willow Way
          24 | Kevin | Carney | 43 Glen Way
          25 | Brian | Waters | 21 Troon Rise
(3 rows)

bpsimple=#
```

Получено три строки, потому что уже был веден клиент из города Lincoln. Однако видно, что данные были добавлены правильно и что создан столбец customer_id. После того как некоторые данные из tcust были загружены в реальную таблицу customer, обычно возникает жела-

ние удалить эти строки из tcust. Для нас же будет удобнее пока оставить все как есть, а удалить их позднее.

Как это работает

Было указано, какие столбцы хотелось бы загрузить в таблицу customer, затем тот же набор данных в том же порядке следования выбран из таблицы tcust. Чтобы предоставить PostgreSQL возможность создания уникальных значений customer_id, столбец customer_id не был включен в список столбцов для загрузки. Поэтому PostgreSQL генерировала уникальные идентификаторы, основываясь на своей последовательности.

Существует и альтернативный способ (он может показаться более простым, особенно если данных для загрузки очень много). Надо ввести во временную таблицу дополнительный столбец, например столбец isvalid типа Boolean. Тогда все данные загружаются во временную таблицу, а все значения isvalid устанавливаются в FALSE посредством оператора UPDATE (формально вы познакомитесь с ним несколько позже):

```
UPDATE tcust SET isvalid = 'FALSE';
```

Инструкция WHERE не задана, следовательно, во всех строках столбец isvalid установлен в FALSE. Проводится работа с данными: для каждой строки, признанной корректной, значение столбца isvalid обновляется и устанавливается в TRUE, затем корректные данные загружаются – для загрузки выбираются только те строки, в которых поле isvalid равно TRUE:

```
bpsimple=# INSERT INTO customer(title, fname, lname, addressline, town,
bpsimple-#  zipcode, phone)
bpsimple-# SELECT title, fname, lname, addressline, town, zipcode, phone
bpsimple-#  FROM tcust WHERE isvalid = TRUE;
```

Как только строки загружены, можно удалить их (подробно об операторе DELETE будет рассказано ближе к концу данной главы) из таблицы tcust; сделаем это так:

```
DELETE FROM tcust WHERE isvalid = TRUE;
```

Потом можно продолжать работать с оставшимися данными таблицы tcust.

Обновление информации в базе данных

Сейчас мы уже знаем, как внести информацию в базу данных посредством оператора INSERT и как извлечь их с помощью SELECT. К сожалению, данные недолго остаются статическими. Люди переезжают, меняют номера телефонов и т. д. Необходимо иметь возможность обновлять данные в базе.

В PostgreSQL, как и во всех СУБД, основанных на SQL, это реализуется при помощи оператора UPDATE.

Оператор UPDATE удивительно прост. Вот его синтаксис:

```
UPDATE tablename SET columnname = value  WHERE condition
```

Для того чтобы одновременно задать несколько столбцов, просто укажем их в списке значений, разделенных запятыми:

```
UPDATE customer SET town = 'Leicester', zipcode = 'LE4 2WQ' WHERE some condition
```

Одновременно можно обновлять сколь угодно много столбцов при условии, что каждый столбец входит в список только один раз. Обратите внимание, что можно использовать только одно имя таблицы, это объясняется особенностями синтаксиса SQL. В тех редких случаях, когда нужно обновить две отдельные, но связанные таблицы, следует написать два отдельных оператора UPDATE. Однако оба эти оператора можно поместить в **транзакцию** (см. главу 9 «Транзакции и блокировки») для того, чтобы убедиться, что оба обновления выполнены или ни одно из них не выполнено.

Попробуйте сами. Оператор UPDATE

Предположим, что наконец удалось выяснить номер телефона для Mr. Bradley в таблице tcust, и теперь хотелось бы обновить данные в настоящей таблице customer. Первая часть оператора UPDATE проста:

```
UPDATE tcust SET phone = '352 3442'
```

Теперь укажем строку для обновления, что также просто:

```
WHERE fname = 'Peter' and lname = 'Bradley';
```

Имея дело с операторами UPDATE, всегда бывает нелишним проверить инструкцию WHERE. Давайте так и сделаем:

```
bpsimple=# SELECT fname, lname, phone FROM tcust WHERE fname = 'Peter' AND
bpsimple-# lname = 'Bradley';
 fname |  lname  | phone
-------+---------+-------
 Peter | Bradley |
(1 row)

bpsimple=#
```

Выбрана единственная строка, которую надо обновить, так что можно спокойно двигаться вперед. Соединим две половинки оператора и выполним его:

```
bpsimple=# UPDATE tcust SET phone = '352 3442' WHERE fname = 'Peter' AND
bpsimple-# lname = 'Bradley';
```

```
UPDATE 1
bpsimple=#
```

PostgreSQL сообщает, что одна строка обновлена. Теперь, если хотите, можно снова выполнить тот же SELECT, чтобы убедиться, что все прошло успешно.

Как это работает

Оператор UPDATE строится в два этапа. Сначала пишем ту часть команды UPDATE, которая будет фактически изменять значение столбца, затем пишем инструкцию WHERE, указывая, какие строки нужно обновить. После проверки WHERE выполняем оператор UPDATE, который изменяет строку предписанным образом.

Предостережение

Почему надо было проявить осторожность и проверить инструкцию WHERE, не исполняя первую часть оператора UPDATE? Дело в том, что оператор UPDATE является абсолютно законным, даже если у него нет секции WHERE. По умолчанию UPDATE в таком случае обновит все строки таблицы, что является желаемым результатом крайне редко.

Таблица tcust — это просто временные данные, используемые для экспериментов, поэтому протестируем на них UPDATE без инструкции WHERE:

```
bpsimple=# UPDATE tcust SET phone = '999 9999';
UPDATE 3
bpsimple=#
```

Как видите, psql сообщает, что три строки были обновлены.

Посмотрим, что получилось:

```
bpsimple=# SELECT fname, lname, phone FROM tcust;
 fname |  lname  |  phone
-------+---------+----------
 Kevin | Carney  | 999 9999
 Brian | Waters  | 999 9999
 Peter | Bradley | 999 9999
(3 rows)

bpsimple=#
```

Так и есть, не этого мы хотели!

Перед исполнением операторов UPDATE ***всегда проверяйте инструкцию*** WHERE**.**

Если требуется обновить несколько строк, то вместо того чтобы извлекать все данные, можно просто проверить, сколько строк соответствуют условию WHERE, используя синтаксис COUNT(*), о котором более подробно будет рассказано в главе 7 «Расширенные возможности

выборки данных». Пока достаточно знать, что если заменить названия столбцов в операторе SELECT на COUNT(*), то будет выведено сообщение о количестве строк, удовлетворяющих условию (вместо возвращения данных строк). На самом деле это практически все, что делает оператор COUNT(*), но на практике он оказывается достаточно полезным.

В качестве примера рассмотрим наш оператор SELECT и проверим, сколько строк соответствуют условию инструкции WHERE:

```
bpsimple=# SELECT count(*) from tcust WHERE fname = 'Peter' AND lname =
bpsimple-# 'Bradley';
 count
-------
     1
(1 row)

bpsimple=#
```

Итак, условие инструкции WHERE ограничено настолько, что задает одну строку. Конечно же, для другого набора данных может оказаться, что задания fname и lname недостаточно для однозначной идентификации строки.

PostgreSQL предоставляет расширение, позволяющее выполнение обновлений из другой таблицы, для этого используется такой синтаксис:

```
UPDATE tablename FROM tablename WHERE condition
```

Это расширение стандарта SQL.

Попробуйте сами. UPDATE с FROM

Для проверки этой возможности создадим таблицу custphone, содержащую фамилии клиентов и номера их телефонов. Таблица будет выглядеть так:

```
create table custphone
(
    customer_id                 serial,
    fname                       varchar(32),
    lname                       varchar(32) not null,
    phone_num                   varchar(16)

);
```

Вставим в нее некоторую информацию о клиентах и их телефонах:

```
bpsimple=# INSERT INTO custphone(fname, lname, phone_num) VALUES('Peter',
bpsimple-# 'Bradley', '352 3442');
INSERT 22593 1

bpsimple=#
```

Теперь необходимо указать в таблице tcust строку, которая должна быть обновлена:

```
bpsimple=# UPDATE tcust SET phone = custphone.phone_num FROM custphone WHERE
bpsimple-# fname = 'Peter' AND lname = 'Bradley';
UPDATE 1

bpsimple=#
```

Как это работает

Создана новая таблица с номерами телефонов клиентов. В нее вставлены данные. В заключение выполнен оператор UPDATE, изменивший строку надлежащим образом.

Оператор UPDATE использует подзапросы для контроля над обновляемыми строками, а FROM разрешает включение столбцов из других таблиц в инструкции SET. На самом деле секция FROM даже не требуется. PostgreSQL по умолчанию создает ссылку на любую таблицу, участвующую в запросе.

Удаление информации из базы данных

Последнее, о чем будет рассказано в этой главе, – это удаление данных из таблиц. Предполагаемые клиенты могут так и не разместить ни одного заказа, заказы могут быть аннулированы и т. д. Поэтому часто приходится удалять информацию из базы данных.

Стандартным способом удаления данных является применение оператора DELETE. Его синтаксис аналогичен оператору UPDATE:

```
DELETE FROM tablename WHERE condition
```

Обратите внимание, что перечисление столбцов отсутствует, поскольку DELETE воздействует на строки. Для удаления данных из столбца применяйте оператор UPDATE, указав для столбца значение NULL или какое-то другое подходящее значение.

Данные о двух новых клиентах скопированы из временной таблицы tcust в постоянную customer, пришло время сделать шаг вперед и удалить эти строки из таблицы tcust.

Попробуйте сами. DELETE

Вам уже известно, какую опасность может таить отсутствие инструкции WHERE в операторах, изменяющих данные. Но случайное удаление данных – это еще более серьезная проблема, поэтому сначала напишем и проверим инструкцию WHERE при помощи оператора SELECT:

```
bpsimple=# SELECT fname, lname FROM tcust WHERE town = 'Lincoln';
 fname | lname
-------+-------
```

```
 Kevin | Carney
 Brian | Waters
(2 rows)

bpsimple=#
```

Все хорошо – извлекаются две строки, которые и ожидались.

Теперь подготовим оператор DELETE и, после последней визуальной проверки, выполним его:

```
bpsimple=# DELETE FROM tcust WHERE town = 'Lincoln';
DELETE 2

bpsimple=#
```

Удалить информацию из базы данных очень просто, будьте внимательны!

Как это работает

Была написана и протестирована инструкция WHERE для выбора предназначенных на удаление строк в базе данных. Выполнен оператор DELETE, который удалил их.

Как и UPDATE, DELETE не может работать с несколькими таблицами одновременно. Если понадобится удалить строки из нескольких таблиц, используйте транзакцию (см. главу 9 «Транзакции и блокировки»).

Существует и другой способ удаления данных из таблицы – оператор TRUNCATE. Он *всегда* удаляет *все* данные из таблицы, и даже если заключить его в транзакцию, все равно восстановить данные не будет никакой возможности. Эту команду следует применять с осторожностью. Синтаксис TRUNCATE таков:

```
TRUNCATE TABLE name of table
```

Используйте данную команду, только если вы абсолютно уверены в том, что все данные таблицы должны быть удалены безвозвратно. В некотором смысле эта операция подобна удалению и пересозданию таблицы, только она гораздо проще и не требует нового определения значения последовательности.

Попробуйте сами. Оператор TRUNCATE

Предположим, что работа с промежуточной таблицей tcust закончена и требуется удалить из нее все данные. Можно с помощью drop удалить таблицу, но в этом случае, если она еще потребуется, придется ее пересоздавать. Вместо этого используем TRUNCATE для удаления всех строк таблицы:

```
bpsimple=# TRUNCATE TABLE tcust;
TRUNCATE

bpsimple=# SELECT COUNT(*) FROM tcust;
```

```
 count
-------
     0
(1 row)

bpsimple=#
```

Все строки удалены.

Как это работает

TRUNCATE просто удаляет все строки указанной таблицы.

> ***Есть два способа удалить все строки из таблицы –*** DELETE ***без инструкции*** WHERE ***и*** TRUNCATE***. Последний, хотя и не включен в SQL-92, является широко распространенным оператором SQL для эффективного удаления всех строк из таблицы.***

Если таблица очень большая (например, несколько тысяч строк) и нужно удалить из нее все строки, то по умолчанию PostgreSQL создаст временную копию данных для восстановления информации в случае отката транзакции. Даже несмотря на то, что в командной строке невозможно явно вызвать транзакцию, все команды автоматически выполняются внутри транзакций. Создание копии нескольких тысяч строк таблицы замедляет исполнение и занимает некоторую часть временного дискового пространства. Оператор TRUNCATE удаляет содержимое таблицы весьма эффективно, при этом резервная копия данных не сохраняется. Поэтому для больших таблиц он действует гораздо более рационально, чем оператор DELETE. Кроме того, даже при использовании внутри транзакции оператор ROLLBACK, который обычно «отменяет» все изменения, не сможет дать обратный ход действиям, произведенным оператором TRUNCATE. Почти всегда лучше придерживаться применения DELETE, т. к. это гораздо более безопасный способ удаления данных. И лишь в редких, специфических случаях, когда важна эффективность и строки должны быть удалены безвозвратно, правильным выбором является TRUNCATE.

Резюме

В данной главе наряду с SELECT были рассмотрены еще три разновидности обработки данных: возможность добавлять данные командой INSERT, модифицировать их с помощью команды UPDATE и, наконец, удалять информацию из базы данных командой DELETE.

Было рассказано о двух формах команды INSERT. Либо данные явно включаются в оператор, либо вставляются данные, выбранные при помощи SELECT из другой таблицы. Показано, что надежнее использовать полный синтаксис оператора INSERT, с перечислением всех столбцов, – в этом случае возникает меньше ошибок. Был представлен родственник команды INSERT, весьма полезное расширение PostgreSQL – коман-

да \copy, позволяющая вставлять данные в таблицу непосредственно из локального файла.

Указано на необходимость осторожного обращения со счетчиками последовательности в полях типа SERIAL, рассказано о том, как проверить значение последовательности и в случае необходимости изменить его. Сделан вывод о том, что лучше предоставить PostgreSQL возможность автоматически генерировать значения последовательности, а не вводить данные для столбцов типа SERIAL.

Рассмотрена работа операторов UPDATE и DELETE, их применение с инструкциями WHERE (аналогично оператору SELECT). Отмечено, что всегда следует проверять операторы UPDATE и DELETE, содержащие WHERE, при помощи SELECT, т. к. ошибки могут привести к трудно поправимым результатам.

Наконец, описан оператор TRUNCATE, являющийся эффективным способом удаления всех строк таблицы. Его необходимо применять очень аккуратно, т. к. удаленные им строки не подлежат восстановлению, даже если он заключен в транзакцию.

7

Расширенные возможности выборки данных

В главе 4 было рассказано об операторе SELECT и о том, как с его помощью извлекать данные. Речь шла о выборе строк, столбцов и объединении таблиц. В этой главе мы рассмотрим оператор SELECT и более подробно изучим его новые возможности. Некоторые из них бывают востребованы очень редко, но знать о них полезно, чтобы иметь четкое представление о том, что возможно в SQL.

Те читатели, которые прорабатывают главу за главой, выполняя все примеры, должны были заметить, что в каждой главе (и в этой в том числе) мы начинаем с «чистых» данных учебной базы данных, поэтому можно обращаться к любым главам, не обязательно соблюдая очередность. Если продолжать работать с данными, полученными из предыдущей главы, то выходная информация будет несколько отличаться. Сценарии из комплекта программ для этой книги позволяют при желании удалить таблицы, создать их заново и заполнить исходными данными.

В этой главе вам встретятся специальные функции, называемые агрегатными, которые позволяют формировать результат на основе данных из нескольких строк. Затем мы изучим объединения, более сложные, чем те, с которыми мы имели дело раньше, они позволят нам управлять результатами. Познакомимся также с новой разновидностью запросов, называемых подзапросами, в которых используются несколько инструкций SELECT. Наконец, поговорим об очень важном внешнем объединении, позволяющем выполнять объединение таблиц более гибко, чем это делалось прежде.

В данной главе мы рассмотрим:

- Агрегатные функции
- Объединения типа UNION
- Подзапросы
- Самообъединения
- Внешние объединения

Агрегатные функции

В предыдущих главах мы уже использовали пару специальных функций: функцию MAX(<имя столбца>), возвращающую наибольшее значение в столбце, и функцию COUNT(*), подсчитывающую количество строк в таблице. Обе они принадлежат к небольшой группе SQL-функций, называемых агрегатными.

В эту группу входят функции:

- COUNT(*)
- COUNT(<имя столбца>)
- MIN(<имя столбца>)
- MAX(<имя столбца>)
- SUM(<имя столбца>)
- AVG(<имя столбца>)

Работать с ними просто, и они часто оказываются весьма полезными.

Встроенная в** psql **команда** \da **выводит все агрегатные функции, используемые PostgreSQL.

COUNT

Начнем с рассмотрения функции COUNT, которая, как это видно из приведенного выше списка, имеет две формы. Функция COUNT(*) подсчитывает количество строк в таблице. Она выступает в роли специального имени столбца в операторе SELECT. Оператор SELECT, в котором используется любая из этих агрегатных функций, может содержать две дополнительные инструкции, GROUP BY и HAVING. В этом случае оператор имеет вид:

```
SELECT COUNT(*) <список столбцов> FROM <таблица> WHERE <условие> [GROUP BY
<столбец> [HAVING <условие агрегирования>]]
```

Необязательная инструкция GROUP BY представляет собой дополнительное условие, накладываемое на оператор SELECT. Обычно она применяется только совместно с агрегатными функциями. GROUP BY может употребляться аналогично инструкции ORDER BY, но применительно к агре-

гатным столбцам. Необязательная инструкция HAVING в составе GROUP BY позволяет выбрать те строки, в которых COUNT(*) удовлетворяет некоторому условию.

Все это выглядит несколько сложным, но в действительности легко применимо на практике. Чтобы понять основную идею, рассмотрим простейший пример использования COUNT(*). Инструкцию GROUP BY рассмотрим чуть позже.

Попробуйте сами. Основы применения COUNT(*)

Предположим, требуется выяснить, сколько в таблице customer клиентов, проживающих в городе Бингаме. Можно, конечно, не мудрствуя лукаво, написать такой SQL-запрос:

```
SELECT * FROM customer WHERE town = 'Bingham';
```

Или, для сокращения объема выходных данных, такой:

```
SELECT customer_id FROM customer WHERE town = 'Bingham';
```

Результат получен, хотя и окольным путем. Большая часть полученных данных нам просто не нужна. Если, предположим, в таблице customer несколько тысяч записей о клиентах, из которых около тысячи живут в Бингаме, объем бесполезных сведений будет достаточно велик. Эту проблему решает функция COUNT(*), позволяющая получить единственную строку, содержащую количество выбранных записей. Оператор SELECT записывается в обычном виде, но вместо реальных столбцов указывается COUNT(*):

```
bpsimple=# SELECT COUNT(*) FROM customer WHERE town = 'Bingham';
 count
-------
     3
(1 row)

bpsimple=#
```

Если необходимо сосчитать всех клиентов, то следует опустить инструкцию WHERE:

```
bpsimple=# SELECT COUNT(*) FROM customer;
 count
-------
    15
(1 row)

bpsimple=#
```

Как видите, получена единственная строка, содержащая искомое значение. Если хотите проверить результат, замените COUNT(*) на customer_id, чтобы получить действительные данные.

Как это работает

Функция COUNT(*) вместо вывода объектов позволяет вывести результат их подсчета. Это намного более эффективно, чем выборка данных, по двум причинам:

- Ненужные данные не извлекаются из базы данных и, что было бы еще хуже, не передаются по сети.
- Функция COUNT(*) позволяет базе данных использовать внутренние сведения о таблице для получения результата, возможно, вовсе не обращаясь к реальным записям.

Никогда не прибегайте к выборке данных, если требуется лишь подсчитать количество строк.

GROUP BY и COUNT(*)

Часто результат, полученный в предыдущем разделе, оказывается достаточным, но не всегда. Предположим, требуется узнать, сколько клиентов проживают в каждом из городов. Это можно выяснить, сделав выборку всех городов, а затем подсчитав количество клиентов в каждом из них. Однако этот путь утомителен и потребует программирования. Не лучше ли выбрать декларативный путь и сформулировать вопрос непосредственно в SQL? Возможно, будет написано нечто вроде:

```
SELECT COUNT(*), town FROM customer;
```

Это предположение, основанное на уже известных вам сведениях, вполне правдоподобно, но PostgreSQL вернет сообщение об ошибке, т. к. выражение синтаксически неверно. Единственное, чего в нем не хватает для решения поставленной задачи – инструкции GROUP BY.

Инструкция GROUP BY сообщает PostgreSQL, что агрегатная функция должна вывести результат, подсчитывая его заново для каждого нового значения указанного столбца (или нескольких столбцов). Это очень просто на практике. Надо лишь добавить конструкцию GROUP BY <имя столбца> в оператор SELECT, содержащий функцию COUNT(*). PostgreSQL сообщит количество строк таблицы, в заданном столбце которых хранится каждое из имеющихся значений.

Попробуйте сами. GROUP BY

Давайте попробуем ответить на вопрос, сколько клиентов живут в каждом из городов?

Прежде всего, напишем оператор SELECT, содержащий функцию COUNT(*) и имя столбца, как в нашем первом предположении:

```
SELECT COUNT(*), town FROM customer;
```

Теперь добавим инструкцию GROUP BY, которая заставит PostgreSQL выводить результат и переустанавливать счетчик каждый раз, когда меняется название города, записав SQL-запрос в таком виде:

```
SELECT COUNT(*), town FROM customer GROUP BY town;
```

Вот как это работает:

```
bpsimple=# SELECT COUNT(*), town FROM customer GROUP BY town;
 count |   town
-------+----------
     3 | Bingham
     1 | Hightown
     1 | Histon
     1 | Lowtown
     1 | Milltown
     2 | Nicetown
     1 | Oahenham
     1 | Oxbridge
     1 | Tibsville
     1 | Welltown
     1 | Winersby
     1 | Yuleville
(12 rows)

bpsimple=#
```

Как видите, получен список городов с указанием количества клиентов в каждом из них.

Как это работает

PostgreSQL упорядочивает результат по значениям в столбце, указанном в инструкции GROUP BY. Затем подсчитывается количество строк, и всякий раз, когда изменяется название города, результат выводится, а счетчик обнуляется. Согласитесь, что это существенно проще, чем писать процедуру, просматривающую в цикле весь список городов.

При желании данный подход можно применить и для нескольких столбцов, если все выбираемые поля указать в инструкции GROUP BY. Предположим, что требуется узнать две вещи. Во-первых, количество клиентов в каждом из городов; во-вторых, сколько различных фамилий они имеют. Просто добавим поле lname в секции SELECT и GROUP BY оператора:

```
bpsimple=# SELECT count(*), lname, town FROM customer GROUP BY town, lname;
 count |  lname  |   town
-------+---------+----------
     1 | Jones   | Bingham
     2 | Stones  | Bingham
     1 | Stones  | Hightown
     1 | Hickman | Histon
     1 | Stones  | Lowtown
```

```
     1 | Hudson  | Milltown
     2 | Matthew | Nicetown
     1 | Cozens  | Oahenham
     1 | Hendy   | Oxbridge
     1 | Howard  | Tibsville
     1 | O'Neill | Welltown
     1 | Neill   | Winersby
     1 | Matthew | Yuleville
(13 rows)

bpsimple=#
```

Обратите внимание, что вывод сортируется сначала по `town`, а затем по `lname`, поскольку именно в таком порядке они перечислены в инструкции `ORDER BY`, и значение «Bingham» теперь встречается дважды – ведь есть два клиента с двумя разными фамилиями, Jones и Stones, которые живут в городе Бингаме.

HAVING и COUNT(*)

Последней необязательной частью оператора является инструкция `HAVING`. Она может смутить человека, впервые столкнувшегося с SQL, но работать с ней совсем несложно. Следует помнить, что инструкция `HAVING` – это аналог `WHERE` для агрегатных функций. `HAVING` применяется для отбора тех строк, для которых значение агрегатной функции удовлетворят некоему условию, например `COUNT(*)>1`. Ее применение совершенно аналогично применению `WHERE` для отбора строк по значению столбца.

> ***Агрегатные функции нельзя использовать в инструкции*** `WHERE`***; они допустимы только в инструкции*** `HAVING`**.**

Рассмотрим пример, из которого все станет ясно. Предположим, нам требуется узнать, в каких городах у нас более одного клиента. Для этого можно воспользоваться функцией `COUNT(*)`, а затем просмотреть получившийся список в поисках подходящих городов. Однако для случая, когда городов несколько тысяч, такое решение не очень подходит. Вместо этого обратимся к инструкции `HAVING` для отбора тех строк, в которых значение `COUNT(*)` больше единицы:

```
bpsimple=# SELECT COUNT(*), town FROM customer
bpsimple-# GROUP BY town HAVING COUNT(*) > 1;
 count |   town
-------+----------
     3 | Bingham
     2 | Nicetown
(2 rows)

bpsimple=#
```

Заметьте, что здесь по-прежнему нужна инструкция `GROUP BY`, которая располагается перед инструкцией `HAVING`. Итак, мы разобрались с осно-

вами применения COUNT(*), GROUP BY и HAVING и теперь соберем все вместе в более сложном примере.

Попробуйте сами. HAVING

Предположим, что требуется составить график поставок. Для этого нам понадобятся сведения о фамилиях и городах клиентов, причем хотелось бы исключить Lincoln (например потому, что это наш собственный город) и выбрать только те города, в которых у нас более одного клиента.

Это не так сложно, как может показаться, надо только разбить решение на несколько шагов. Такой подход часто оправдывает себя в SQL. Если задача кажется чересчур сложной, начните с решения более простой, но аналогичной задачи, а затем уточняйте решение, пока не получите искомый результат. Другими словами, возьмите проблему, поделите ее на части и каждую из них решайте отдельно.

Начнем с простой выборки данных без подсчета:

```
bpsimple=# SELECT lname, town FROM customer WHERE town <> 'Lincoln';
  lname  |   town
---------+-----------
 Stones  | Hightown
 Stones  | Lowtown
 Matthew | Nicetown
 Matthew | Yuleville
 Cozens  | Oahenham
 Matthew | Nicetown
 Stones  | Bingham
 Stones  | Bingham
 Hickman | Histon
 Howard  | Tibsville
 Jones   | Bingham
 Neill   | Winersby
 Hendy   | Oxbridge
 O'Neill | Welltown
 Hudson  | Milltown
 Smyth   | Milltown
 Garrett | Lowtown
(17 rows)

bpsimple=#
```

Пока все хорошо, не так ли?

Теперь, если использовать COUNT(*) для подсчета, то придется сгруппировать результаты по полям lname и town:

```
bpsimple=# SELECT COUNT(*), lname, town FROM customer WHERE town <>
bpsimple-# 'Lincoln' GROUP BY lname, town;
```

```
 count |  lname  |   town
-------+---------+-----------
     1 | Cozens  | Oahenham
     1 | Garrett | Lowtown
     1 | Hendy   | Oxbridge
     1 | Hickman | Histon
     1 | Howard  | Tibsville
     1 | Hudson  | Milltown
     1 | Jones   | Bingham
     2 | Matthew | Nicetown
     1 | Matthew | Yuleville
     1 | Neill   | Winersby
     1 | O'Neill | Welltown
     1 | Smyth   | Milltown
     2 | Stones  | Bingham
     1 | Stones  | Hightown
     1 | Stones  | Lowtown
(15 rows)

bpsimple=#
```

Уже можно получить ответ, просто просматривая данные, но для получения окончательного решения осталось лишь добавить инструкцию `HAVING`, которая отберет строки со значением `COUNT(*)`, большим единицы:

```
bpsimple=# SELECT COUNT(*), lname, town FROM customer WHERE town <>
bpsimple-# 'Lincoln' GROUP BY lname, town HAVING COUNT(*) > 1;
 count |  lname  |   town
-------+---------+----------
     2 | Matthew | Nicetown
     2 | Stones  | Bingham
(2 rows)

bpsimple=#
```

Ничего сложного, если поделить задачу на части.

Как это работает

Решение получено в три этапа:

- Написан постой оператор `SELECT`, выбирающий все интересующие нас строки.
- Добавлены `COUNT(*)` и `GROUP BY` для подсчета неповторяющихся сочетаний `lname` и `town`.
- И наконец, добавлена инструкция `HAVING`, отбирающая только те строки, в которых значение `COUNT(*)` больше единицы.

Такой подход, однако, таит в себе одну «маленькую неприятность». При работе с таблицей, содержащей несколько тысяч записей, прокрутка длинного списка клиентов на экране в процессе разработки

запроса может занимать значительное время. Для нашей учебной базы это не проблема, но при большом объеме данных такой способ имеет некоторые недостатки. К счастью, почти всегда есть возможность применить первичный ключ для разработки запросов на фрагментах данных. Если во всех наших запросах использовать условие `WHERE customer_id < 50`, то можно будет работать с подмножеством из первых пятидесяти клиентов.

Закончив разработку, достаточно просто убрать условие `WHERE`, чтобы распространить найденное решение на всю таблицу. Разумеется, следует помнить, что выбранное при разработке подмножество данных может оказаться нерепрезентативным, и на нем не всегда можно провести исчерпывающее тестирование запроса.

COUNT(<имя столбца>)

Другая разновидность функции `COUNT(*)` используется при замене `'*'` на имя столбца. Отличие заключается в том, что `COUNT(<имя столбца>)` подсчитывает те вхождения строк, где указанный столбец имеет значение, отличное от `NULL`.

Предположим, что в таблицу `customer` добавлены данные о новых клиентах, при этом некоторые записи содержат `NULL` в поле номера телефона:

```
INSERT INTO customer(title, fname, lname, addressline, town, zipcode)
VALUES('Mr','Gavin','Smyth','23 Harlestone','Milltown','MT7 7HI');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mrs','Sarah','Harvey','84 Willow Way','Lincoln','LC3 7RD','527 3739');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode)
VALUES('Mr','Steve','Harvey','84 Willow Way','Lincoln','LC3 7RD');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode)
VALUES('Mr','Paul','Garrett','27 Chase Avenue','Lowtown','LT5 8TQ');
```

Посмотрим, сколько у нас клиентов с неизвестными телефонными номерами:

```
bpsimple=# SELECT customer_id FROM customer WHERE phone IS NULL;
 customer_id
-------------
          16
          18
          19
(3 rows)

bpsimple=#
```

Как видите, номера телефонов отсутствуют у трех клиентов. Посмотрим, сколько клиентов всего:

```
bpsimple=# SELECT COUNT(*) FROM customer;
 count
---------
```

```
    19
(1 row)

bpsimple=#
```

Всего 19 клиентов. Если теперь подсчитать количество тех из них, для которых значение столбца phone отлично от NULL, то результат, скорее всего, будет равен 16:

```
bpsimple=# SELECT COUNT(phone) FROM customer;
 count
---------
    16
(1 row)

bpsimple=#
```

В этом заключается единственное различие между функциями COUNT(*) и COUNT(column name). В варианте с явным указанием столбца подсчитываются только те строки, в которых указанный столбец отличен от NULL, в то время как вариант с использованием '*' учитывает все строки. Во всех прочих случаях, таких как применение в GROUP BY и HAVING, этот вариант работает в точности как и COUNT(*).

Функция MIN()

Итак, мы рассмотрели функцию COUNT(*) и усвоили правила работы с ней. Теперь можно применить аналогичный подход и ко всем остальным агрегатным функциям.

Как и следовало ожидать, MIN() получает в качестве параметра имя столбца и возвращает наименьшее из значений, найденных в этом столбце. Для столбцов числового типа результат очевиден. Для данных типа даты (и времени) возвращается самая ранняя из дат, которые могут относиться как к прошлому, так и к будущему. Для строк переменной длины результат оказывается немного неожиданным; строки сравниваются после дополнения пробелами справа. Применяйте функции MIN() или MAX() для столбцов типа VARCHAR с осторожностью, результат может отличаться от ожидаемого.

Приведем два примера.

Найти наименьшую стоимость доставки, включенную в заказ:

```
bpsimple=# SELECT MIN(shipping) FROM orderinfo;
 min
-------
 0.00
(1 row)

bpsimple=#
```

Получена нулевая стоимость. Посмотрим, что произойдет, если применить эту функцию к столбцу телефонных номеров, где, как известно, имеются значения NULL:

```
bpsimple=# SELECT MIN(phone) FROM customer;
   min
-----------
 010 4567
(1 row)

bpsimple=#
```

Можно было бы ожидать, что результат будет иметь значение NULL или пустой сроки. Но, принимая во внимание тот факт, что NULL представляет собой неопределенное значение, функция MIN() игнорирует такие строки. Игнорирование значений NULL характерно для всех агрегатных функций за исключением COUNT(*). Насколько ценным является знание наименьшего из телефонных номеров – это уже другой вопрос.

Функция MAX()

Нет ничего удивительного в том, что функция MAX() похожа на MIN(), но выполняет противоположные действия.

Как и следовало ожидать, MAX() принимает имя столбца в качестве параметра и возвращает наибольшее значение, найденное в этом столбце.

Приведем два примера.

Найти наибольшую стоимость доставки, включенную в заказ:

```
bpsimple=# SELECT MAX(shipping) FROM orderinfo;
 max
-------
 3.99
(1 row)

bpsimple=#
```

Как и в MIN(), значения NULL игнорируются:

```
bpsimple=# SELECT MAX(phone) FROM customer;
   max
-----------
 961 4526
(1 row)

bpsimple=#
```

Пожалуй, это все, что следует знать о функции MAX() за исключением того, что в инструкциях GROUP BY и HAVING она применяется аналогично COUNT(*).

Функция SUM()

Функция SUM() принимает в качестве параметра имя числового столбца и возвращает итоговое значение. Как и в MIN() и MAX(), значения NULL игнорируются:

```
bpsimple=# SELECT SUM(shipping) FROM orderinfo;
 sum
-------
 9.97
(1 row)

bpsimple=#
```

Есть, однако, одна интересная разновидность функции SUM(). Можно попросить ее подсчитать сумму неповторяющихся значений – так, чтобы строки, имеющие одинаковые значения, были учтены лишь единожды:

```
bpsimple=# SELECT SUM(DISTINCT shipping) FROM orderinfo;
 sum
-------
 6.98
(1 row)

bpsimple=#
```

Надо признать, что непросто представить себе причину использования этого варианта функции в реальном мире.

Функция AVG()

Последняя из агрегатных функций – AVG(). Она тоже принимает имя столбца и возвращает его среднее значение. Как и SUM(), она игнорирует значения NULL и также может принимать ключевое слово DISTINCT для работы с неповторяющимися значениями:

```
bpsimple=# SELECT AVG(shipping) FROM orderinfo;
      avg
---------------
 1.9940000000
(1 row)

bpsimple=#
```

Вариант с применением DISTINCT:

```
bpsimple=# SELECT AVG(DISTINCT shipping) FROM orderinfo;
      avg
--------------
 2.3266666667
(1 row)

bpsimple=#
```

Обратите внимание, что и в стандартном SQL, и в его реализации в PostgreSQL отсутствуют функции MODE или MEDIAN, хотя некоторые поставщики включают их в свои коммерческие продукты в качестве расширения.

Объединение UNION

Посмотрим теперь, как объединить несколько операторов SELECT для построения более сложных запросов.

Вспомните, что в предыдущей главе таблица tcust использовалась в качестве вспомогательной, когда добавлялись данные в основную таблицу customer. Предположим, что в промежуток времени после загрузки таблицы tcust новыми данными о клиентах и до ее очистки и загрузки в основную таблицу customer возникла необходимость вывести список всех городов, где есть клиенты, в том числе и новые. Разумно было бы заметить, что новые данные о клиентах еще не отредактированы и не загружены в основную таблицу, поэтому нет уверенности в правильности новых данных, и любой список городов, объединяющий два списка, также не будет точным. Это вполне возможно, но не важно. Вероятно, значение имеют только общие показатели географического разброса клиентов, а не точные данные.

Можно было бы решить задачу, выбрав столбец town из таблицы customer, сохранив его, а затем выбрать town из таблицы tcust, опять-таки сохранить его, а затем соединить два списка. Однако такое решение выглядит громоздко, поскольку в обеих таблицах есть списки городов.

Нет ли какого-нибудь способа объединить списки? По названию раздела можно было догадаться, что такой способ есть и называется он объединением UNION. Такие объединения используются нечасто, но при некоторых обстоятельствах это именно то, что нужно, чтобы решить задачу, к тому же работать с ними несложно.

Поместим в таблицу tcust данные, чтобы она выглядела так:

```
bpsimple=# SELECT * FROM tcust;
title|  fname  |  lname   |  addressline   |   town   |  zipcode   |  phone
-----+---------+----------+----------------+----------+------------+--------
 Mr  | Peter   | Bradley  | 72 Milton Rise | Keynes   | MK41 2HQ   |
 Mr  | Kevin   | Carney   | 43 Glen Way    | Lincoln  | LI2 7RD    | 786 3454
 Mr  | Brian   | Waters   | 21 Troon Rise  | Lincoln  | LI7 6GT    | 786 7245
 Mr  | Malcolm | Whalley  | 3 Craddock Way | Welltown | WT3 4GQ    | 435 6543
(4 rows)

bpsimple=#
```

Сравним запрос списка городов из этой таблицы и таблицы customer.

Нам известно, как выбрать столбец town из каждой таблицы. Напишем пару очень простых операторов SELECT:

```
SELECT town FROM tcust;
SELECT town FROM customer;
```

Каждый из них выводит список городов. Чтобы объединить их, используем оператор UNION, «сшивающий» результаты двух операторов SELECT:

```
SELECT town FROM tcust UNION SELECT town FROM customer;
```

Попробуйте сами. Объединение UNION

Вводим оператор SQL, разбив его на несколько строк для удобства чтения. Заметьте, что приглашение psql на ввод команды меняется с =# на –#, чтобы показать, что это строка продолжения и что точка с запятой встречается всего один раз, в самом конце, т. к. все это – единый оператор:

```
bpsimple=# SELECT town FROM tcust
bpsimple-# UNION
bpsimple-# SELECT town FROM customer;
   town
------------
 Bingham
 Hightown
 Histon
 Keynes
 Lincoln
 Lowtown
 Milltown
 Nicetown
 Oahenham
 Oxbridge
 Tibsville
 Welltown
 Winersby
 Yuleville
(14 rows)

bpsimple=#
```

Как это работает

PostgreSQL берет списки городов из обеих таблиц и объединяет их в один. К тому же, она удаляет все повторения. Если же требуется получить список всех городов, содержащий повторения, можно написать UNION ALL, а не просто UNION.

Возможность комбинировать операторы SELECT не ограничена одним столбцом, можно объединить города и почтовые индексы:

```
SELECT town, zipcode FROM tcust UNION SELECT town, zipcode FROM customer;
```

Мы получим список, в котором будут присутствовать оба столбца. Такой список будет более длинным, поскольку в него включен почтовый индекс, следовательно, будет извлечено больше уникальных строк.

И все-таки объединение UNION – не волшебник. Два списка столбцов из двух таблиц, которые необходимо объединить, должны иметь одинаковое количество столбцов, а выбранные столбцы должны иметь совместимые типы. Давайте рассмотрим:

```
bpsimple=# SELECT title FROM customer
bpsimple-# UNION
bpsimple-# SELECT town FROM tcust;
  title
----------
 Keynes
 Lincoln
 Miss
 Mr
 Mrs
 Welltown
(6 rows)

bpsimple=#
```

Этот запрос, хотя и не имеет никакого смысла, но вполне законен, т. к. PostgreSQL может объединить столбцы, даже несмотря на то, что title является строкой фиксированной длины, а town – строкой переменной длины, по крайней мере, оба значения представляют собой символьные строки. Если же попытаться, например, объединить столбцы customer_id и town, то PostgreSQL сообщит, что такая операция невозможна, поскольку типы столбцов различны.

В целом это все, что следует знать об объединениях UNION, которые могут быть очень удобным способом комбинирования данных из двух или более таблиц.

Подзапросы

После знакомства с командами SQL, содержащими в себе несколько операторов SELECT, можно обратиться к целому классу операторов извлечения данных, которые гораздо более сложными способами объединяют несколько операторов SELECT. Понять их несколько сложнее, чем отдельные запросы SELECT или объединения UNION, но они весьма полезны и делают доступной новую область критериев выборки данных.

Подзапрос – это выражение, в котором одно (или несколько) из условий WHERE оператора SELECT также является оператором SELECT.

Предположим, что нужно найти в таблице item все продукты, себестоимость (cost price) которых превышает 10 долларов. Оператор SELECT несколько усложняется необходимостью приведения числа, с которым производится сравнение, к типу NUMERIC(7,2), чтобы оно было совместимо по типу со значениями столбца cost_price таблицы item, но по существу он все так же прост:

```
bpsimple=# SELECT * FROM item WHERE cost_price > CAST(10.0 AS NUMERIC(7,2));
 item_id | description | cost_price | sell_price
---------+-------------+------------+------------
       1 | Wood Puzzle |      15.23 |      21.95
       7 | Fan Large   |      13.36 |      19.95
      11 | Speakers    |      19.73 |      25.32
(3 rows)

bpsimple=#
```

Предположим, что необходимо найти все товары, себестоимость которых превышает среднюю себестоимость. Это несложно сделать при помощи двух запросов:

```
bpsimple=# SELECT AVG(cost_price) FROM item;
     avg
----------------
 7.2490909091
(1 row)

bpsimple=# SELECT * FROM item WHERE cost_price > CAST(7.249 AS
bpsimple-# NUMERIC(7,2));
 item_id |  description  | cost_price | sell_price
---------+---------------+------------+------------
       1 | Wood Puzzle   |      15.23 |      21.95
       2 | Rubic Cube    |       7.45 |      11.49
       5 | Picture Frame |       7.54 |       9.95
       6 | Fan Small     |       9.23 |      15.75
       7 | Fan Large     |      13.36 |      19.95
      11 | Speakers      |      19.73 |      25.32
(6 rows)

bpsimple=#
```

Но такое решение некрасиво. На самом деле хотелось бы передать результат первого запроса прямо во второй, чтобы не надо было запоминать его, а затем заново вводить для второго запроса.

В разрешении такой ситуации помогают подзапросы. Заключим первый запрос в скобки и используем его как часть инструкции WHERE второго запроса:

```
bpsimple=# SELECT * FROM item WHERE cost_price > (SELECT AVG(cost_price)
bpsimple-# FROM item);
 item_id |  description  | cost_price | sell_price
---------+---------------+------------+------------
       1 | Wood Puzzle   |      15.23 |      21.95
       2 | Rubic Cube    |       7.45 |      11.49
       5 | Picture Frame |       7.54 |       9.95
       6 | Fan Small     |       9.23 |      15.75
       7 | Fan Large     |      13.36 |      19.95
      11 | Speakers      |      19.73 |      25.32
(6 rows)

bpsimple=#
```

Как видите, получен тот же самый результат, но ни промежуточный шаг, ни приведение типа не понадобились, т. к. результат уже принадлежит к правильному типу.

Первым PostgreSQL выполняет запрос, стоящий в скобках. После получения ответа запускается второй запрос, в который подставляется результат внутреннего запроса. При желании можно написать несколько подзапросов, используя различные инструкции `WHERE`. Не обязательно ограничиваться одним, но необходимость в нескольких вложенных операторах `SELECT` возникает редко.

Попробуйте сами. Подзапросы

Рассмотрим более сложный пример. Пусть требуется найти все товары, себестоимость которых выше ее среднего значения, а цена продажи ниже среднего значения для цены продажи. Вероятно, это должно наводить на мысль о том, что наша прибыль не очень велика, поэтому будем надеяться, что критерию удовлетворяют не слишком много товаров.

Мы уже знаем, как вычислить среднюю себестоимость: `SELECT AVG(cost_price) FROM item`. Нахождение средней цены продажи выполняется аналогично: `SELECT AVG(sell_price) FROM item`.

Основной запрос будет иметь такой вид:

```
SELECT * FROM item WHERE cost_price > average cost price AND sell_price <
average selling price
```

Если сложить эти три запроса, получим:

```
bpsimple=# SELECT * FROM item WHERE cost_price > (SELECT AVG(cost_price)
bpsimple-# FROM item) AND sell_price < (SELECT AVG(sell_price) FROM item);
 item_id |  description  | cost_price | sell_price
---------+---------------+------------+------------
       5 | Picture Frame |       7.54 |       9.95
(1 row)

bpsimple=#
```

Похоже, кому-то хочется посмотреть на цены рамок для картин и проверить их корректность!

Как это работает

PostgreSQL сначала просматривает запрос и обнаруживает два запроса в скобках, т. е. два подзапроса. При этом каждый запрос оценивается независимо, а затем помещает результаты обратно в соответствующую часть инструкции `WHERE` основного запроса, прежде чем приступать к его выполнению.

Также можно было прибегнуть к дополнительной инструкции `WHERE` или `ORDER BY`. Сочетание условий `WHERE`, определяемых подзапросами, с более стандартными условиями абсолютно законно.

Типы подзапросов

Пока что рассматривались только подзапросы, возвращавшие один результат, т. к. в них применялась агрегатная функция. Вообще подзапросы могут иметь три типа результатов:

- одно значение (как нам уже встречавшиеся)
- ноль или более строк
- результат проверки существования данных

Поговорим о втором типе подзапроса с возможностью возврата нескольких строк. Предположим, что требуется узнать, себестоимость каких товаров, имеющихся на складе, превышает 10,0. Мы уже умеем решать такую задачу при помощи одного оператора `SELECT`:

```
bpsimple=# SELECT s.item_id, s.quantity FROM stock s, item i WHERE
bpsimple-# i.cost_price > CAST(10.0 AS NUMERIC(7,2)) AND s.item_id =
bpsimple-# i.item_id;
 item_id | quantity
---------+----------
       1 |       12
       7 |        8
(2 rows)

bpsimple=#
```

Обратите внимание, для того чтобы сделать запрос короче, использованы псевдонимы для таблиц (`stock` превращается в `s`, `item` – в `i`). Надо объединить две таблицы (`s.item_id = i.item_id`), добавив условие для себестоимости в таблице `item` (`i.cost_price > CAST(10.0 AS NUMERIC(7,2))`).

Можно оформить это и как подзапрос, указав ключевое слово `IN` для проверки списка значений. Надо написать запрос, выдающий список идентификаторов (`item_id`) тех товаров, себестоимость которых больше 10,0:

```
SELECT item_id FROM item WHERE cost_price > CAST(10.0 AS NUMERIC(7,2));
```

Также необходим запрос, выбирающий товары из таблицы `stock`:

```
SELECT * FROM stock WHERE item_id IN list of values
```

Теперь можно соединить два запроса:

```
bpsimple=# SELECT * FROM stock WHERE item_id IN (SELECT item_id FROM item
bpsimple-# WHERE cost_price > CAST(10.0 AS NUMERIC(7,2)));
 item_id | quantity
---------+----------
       1 |       12
       7 |        8
(2 rows)

bpsimple=#
```

Получаем тот же самый результат. Часто бывает так, что подзапросы можно переписать как объединения, но все же это верно не для всех запросов, поэтому необходимо разобраться в них. Как и во многих обычных запросах, можно инвертировать условие, написав `NOT IN`, а можно также вводить дополнительные инструкции `WHERE` и `ORDER BY`.

Если подзапрос может быть переписан в виде объединения, то какой из двух вариантов выбрать? Следует принять во внимание две вещи: удобочитаемость и производительность. Если запрос время от времени выполняется для небольших таблиц и выполняется быстро, то используйте ту форму, которую считаете более удобной для чтения. Если же запрос интенсивно используется для больших таблиц, то стоит написать его разными способами и посмотреть, какой из них будет иметь лучшую производительность. Может оказаться, что благодаря оптимизатору запросов производительность во всех случаях будет одинаковой, тогда автоматически победит удобочитаемость.

> ***Проверяя производительность операторов SQL, имейте в виду, что вам неподвластны многие параметры, например кэширование данных операционной системой.***

Может оказаться, что решающее влияние на производительность оказывает тип данных, содержащихся в базе, или же количество строк в таблицах.

Еще не был рассмотрен третий тип подзапросов – проверки на существование. Дело в том, что он достаточно сложен, но не волнуйтесь, мы вернемся к нему до того, как закончится эта глава.

Связанные подзапросы

До сих пор обсуждались типы подзапросов, в которых запрос выполняется для получения ответа, который затем вставляется во второй запрос. Однако никак иначе два запроса между собой не связаны и назы-

ваются несвязанными подзапросами (uncorrelated subqueries). Между внутренним и внешним запросами нет связующей таблицы. Может случиться так, что один и тот же столбец одной и той же таблицы фигурирует в обеих частях оператора SELECT, но связаны они только результатом подзапроса, возвращенным в инструкцию WHERE основного запроса.

Есть и другая группа подзапросов, называемых связанными подзапросами (correlated subqueries), в которых отношение между двумя частями запроса является гораздо более сложным. В связанном подзапросе таблица во внутреннем операторе SELECT объединяется с таблицей внешнего SELECT, поэтому запросы и называют связанными. Существует многочисленная группа подзапросов, которые часто не могут быть переписаны в виде простых операторов SELECT с объединениями.

Общая форма связанного запроса выглядит так:

```
SELECT columnA FROM table1 T1 WHERE T1.columnB = (SELECT T2.columnB FROM
table2 T2 WHERE T2.columnC = T1.columnC)
```

Чтобы упростить объяснение, мы использовали псевдо-SQL. Необходимо обратить внимание на то, что таблица T1 из внешнего оператора SELECT появляется также и во внутреннем операторе SELECT. Следовательно, внешний и внутренний запросы считаются связанными. Вы должны были заметить, что названия таблиц заменяются псевдонимами. Это важно, поскольку правила, касающиеся названий таблиц в связанных запросах, достаточно сложны, и даже небольшая ошибка может привести к неожиданным результатам.

Настоятельно рекомендуем использовать псевдонимы для названий всех таблиц в связанном запросе, т. к. это наиболее надежный вариант.

При выполнении такого запроса имеет место достаточно сложная последовательность действий. Сначала строка таблицы T1 извлекается для внешнего оператора SELECT, затем столбец T1.columnB передается внутреннему запросу, выполняющему выборку из таблицы T2 на основании переданной ему информации. Потом результат передается обратно во внешний запрос, который оценивает инструкцию WHERE, прежде чем перейти к следующей строке.

Процесс представлен на рис. 7.1:

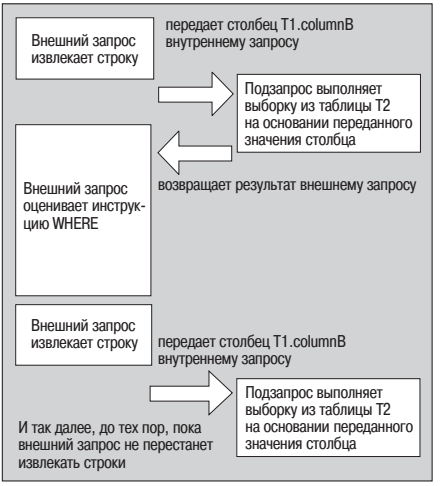


Рис. 7.1. Выполнение связанного подзапроса

Выглядит довольно запутанно, но это так и есть. Связанные подзапросы часто выполняются неэффективно. Однако в некоторых случаях они могут решить очень сложные задачи. Поэтому и полезно знать об их существовании, даже несмотря на то, что применять их вы будете не часто.

Попробуйте сами. Связанный подзапрос

Для простой базы данных, вроде той, с которой мы работаем, нет необходимости в применении связанных запросов. Как правило, их можно записать другими способами, но все же попробуем продемонстрировать работу связанных подзапросов на примере учебной базы данных.

Пусть требуется узнать дату размещения заказов клиентов из города Bingham. Можно было бы сделать это более традиционным способом, но в данном случае используем связанный подзапрос:

```
bpsimple=# SELECT oi.date_placed FROM orderinfo oi WHERE oi.customer_id =
bpsimple-# (SELECT c.customer_id from customer c WHERE c.customer_id =
bpsimple-# oi.customer_id and town = 'Bingham');
 date_placed
-------------
 2000-06-23
 2000-07-21
(2 rows)

bpsimple=#
```

Как это работает

Выполнение запроса начинается с выбора строки из таблицы `orderinfo`. Затем выполняется подзапрос для таблицы `customer` на основании полученного `customer_id`. Подзапрос просматривает строки таблицы `customer` в поиске той, в которой `customer_id` из внешнего запроса соответствует значению города «Bingham». Если он ее находит, то передает `customer_id` обратно в исходный запрос, который оценивает выражение `WHERE` и, если оно истинно, то выводит столбец `date_placed`. Внешний запрос принимается за следующую строку, и последовательность действий повторяется.

Рассмотрим другой пример. На этот раз применим третий тип подзапроса, который вам еще не встречался. А именно тот, где подзапрос проверяет наличие данных.

Предположим, что требуется вывести список всех клиентов, делавших заказы. В учебной базе данных их немного. Первую часть запроса написать просто:

```
SELECT fname, lname FROM customer c;
```

Отметьте, что название таблицы customer заменено псевдонимом c и готово к использованию в подзапросе. В следующей части запроса необходимо определить, присутствует ли customer_id и в таблице orderinfo:

```
SELECT 1 FROM orderinfo oi WHERE oi.customer_id = c.customer_id;
```

Необходимо обратить внимание на два важных момента. Во-первых, мы прибегли к общепринятому «трюку». Для того чтобы выполнить запрос, результаты которого не важны, просто пишем на месте имени столбца '1'. То есть если какие-то данные будут обнаружены, то будет возвращена 1. Это простой и эффективный способ получить значение «истина». Выглядит достаточно таинственно, так что давайте попробуем выполнить:

```
bpsimple=# SELECT 1 FROM customer WHERE town = 'Bingham';
 ?column?
----------
        1
        1
        1
(3 rows)

bpsimple=#
```

Такой способ может показаться немного странным, но он работает. Важно не применять в данном случае функцию COUNT(*), поскольку надо не просто узнать, сколько клиентов живут в городе Бингеме, а получить результат из каждой строки, значением города в которой является «Bingham».

Вторым важным моментом является использование в этом подзапросе столбца customer, присутствовавшего в исходном запросе. Так они связываются между собой. Как и раньше, вместо названий таблиц указываем псевдонимы. Теперь надо соединить две половинки.

Пришло время познакомиться с последней формой подзапроса, через которую мы перескочили ранее. Подзапрос проверяет существование данных с помощью ключевого слова EXISTS в инструкции WHERE, при этом ему не обязательно знать, какие конкретно данные имеются.

В нашем запросе применение EXISTS представляет собою хороший способ объединения двух операторов SELECT, т. к. необходимо знать лишь, возвращает ли подзапрос строку. Инструкция EXISTS обычно работает более эффективно, чем другие виды объединений и условия IN. Поэтому во многих случаях, когда есть несколько способов записать подзапрос, стоит предпочесть именно этот вариант.

```
bpsimple=# SELECT fname, lname FROM customer c WHERE EXISTS ( SELECT 1 FROM
bpsimple-# orderinfo oi WHERE oi.customer_id = c.customer_id);
```

```
 fname |  lname
-------+---------
 Alex  | Matthew
 Ann   | Stones
 Laura | Hendy
 David | Hudson
(4 rows)

bpsimple=#
```

Итак, вы видели, как пишутся связанные подзапросы. Если какую-то ситуацию не удастся разрешить обычными запросами, то может оказаться, что именно этот способ поможет решить проблему.

Самообъединения

Существует особый вид объединений, называемый самообъединением (self join), который применяется, когда нужно объединить столбцы одной таблицы. Такая потребность возникает нечасто, но в каких-то случаях самообъединения могут быть очень полезны, поэтому представим их краткое описание.

Предположим, что продаются товары, которые можно продавать как по отдельности, так и в наборе. Пусть, например, продаются стол и стул по отдельности, а также набор стульев вместе со столом. Хотелось бы хранить не только отдельные товары, но и отношения между ними при продаже в составе одного набора. Такая ситуация часто называется «дроблением номенклатуры» (parts explosion), она еще встретится нам в главе 12.

Начнем с создания таблицы, ькоторая может хранить не только идентификатор товара и его описание, но и второй идентификатор товара:

```
CREATE TABLE part (part_id INT, description VARCHAR(32), parent_part_id INT);
```

Используем `parent_part_id` для хранения идентификатора того, компонентом чего является товар. Например, пусть есть набор из стола и стульев (`item_id 1`), который состоит из стульев (`item_id 2`) и стола (`item_id 3`). Операторы `INSERT` будут выглядеть так:

```
bpsimple=# INSERT INTO part(part_id, description, parent_part_id) VALUES(1,
bpsimple-# 'table and chairs', NULL);
INSERT 21579 1
bpsimple=# INSERT INTO part(part_id, description, parent_part_id) VALUES(2,
bpsimple-# 'chair', 1);
INSERT 21580 1

bpsimple=# INSERT INTO part(part_id, description, parent_part_id) VALUES(3,
```

```
bpsimple-# 'table', 1);
INSERT 21581 1

bpsimple=#
```

Теперь данные сохранены, но как извлечь информацию о том, из каких частей состоит определенный компонент? Необходимо объединить части таблицы с ней самой.

Это достаточно просто. Заменим псевдонимами названия таблиц и напишем инструкцию WHERE, ссылающуюся на ту же самую таблицу, только с другим именем:

```
bpsimple=# SELECT p1.description, p2.description FROM part p1, part p2 WHERE
bpsimple-# p1.part_id = p2.parent_part_id;
   description    | description
------------------+-------------
 table and chairs | chair
 table and chairs | table
(2 rows)

bpsimple=#
```

Все работает. Несколько смущает лишь то, что два выходных столбца называются одинаково. Ситуацию легко исправить, дав им имена при помощи AS:

```
bpsimple=# SELECT p1.description AS "Combined", p2.description AS "Parts"
bpsimple-# FROM part p1, part p2 WHERE p1.part_id = p2.parent_part_id;
     Combined     | Parts
------------------+-------
 table and chairs | chair
 table and chairs | table
(2 rows)

bpsimple=#
```

Еще раз поговорим о самообъединениях в главе 12, когда будем обсуждать возможность хранения в одной таблице отношений между начальниками и подчиненными.

Внешние объединения

Последней важной темой, рассмотренной в данной главе, будут так называемые внешние объединения (outer joins). Они похожи на обычные объединения, но в них применяется несколько другой синтаксис, поэтому знакомство с ними и было отложено до конца главы.

Посмотрим на таблицы item и stock (рис. 7.2):

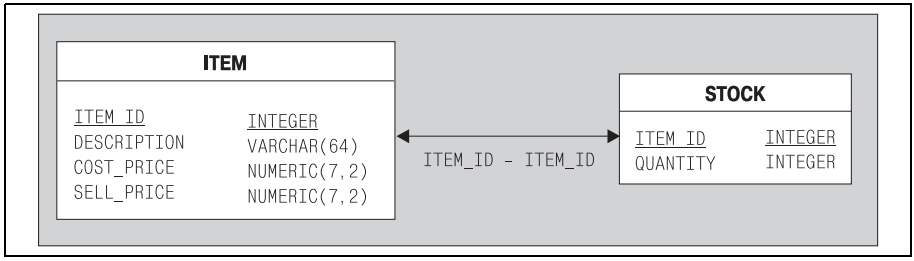


Рис. 7.2. Таблицы item и stock

Если помните, все товары, предлагаемые на продажу, хранятся в таблице item, а в таблице stock представлены только те из них, которые имеются в наличии на складе.

Предположим, что требуется получить список всех продаваемых продуктов с указанием их количества, хранящегося на складе. Этот на первый взгляд простой запрос, оказывается, очень трудно (хотя и возможно) написать на уже известном нам SQL. Весьма поучительно будет найти такое решение, поэтому решим задачу, применяя только уже известный SQL.

Попробуем выполнить простой оператор SELECT, объединяя две таблицы:

```
bpsimple=# SELECT i.item_id, s.quantity FROM item i, stock s WHERE i.item_id
bpsimple-# = s.item_id;
 item_id | quantity
---------+----------
       1 |       12
       2 |        2
       4 |        8
       5 |        3
       7 |        8
       8 |       18
      10 |        1
(7 rows)

bpsimple=#
```

Легко заметить (т. к. нам известно, что идентификаторы товаров – item_id в таблице item – представляют собой последовательные значения, без пропусков), что некоторые item_id пропущены. Пропущены строки, связанные с товарами, которых нет на складе, поскольку объединение таблиц item и stock для этих строк не работает, ведь в таблице stock нет записи для такого item_id.

Можно найти недостающие строки, применяя подзапрос и инструкцию IN:

```
bpsimple=# SELECT i.item_id FROM item i WHERE i.item_id NOT IN (SELECT
bpsimple-# i.item_id FROM item i, stock s WHERE i.item_id = s.item_id);
```

```
 item_id
---------
       3
       6
       9
      11
(4 rows)

bpsimple=#
```

Пересказать это на обычном языке можно так: «выбрать из таблицы item все item_id, кроме тех, что встречаются и в таблице stock».

Внутренний оператор SELECT уже рассматривался, но на этот раз мы используем возвращенный им список item_id как часть другого оператора SELECT. Основной оператор SELECT перечисляет все известные item_id, за исключением удаляемых инструкцией WHERE NOT IN идентификаторов, выведенных подзапросом.

Теперь у нас есть список item_id товаров, которых нет на складе, и список item_id, запас которых есть на складе, но получены они при помощи разных запросов. Теперь надо объединить два списка вместе, чем и займется UNION. Однако есть небольшая трудность. Первый оператор возвращает два столбца: item_id и quantity, а второй SELECT — только один, item_id, т. к. запаса этих товаров нет. Необходимо добавить фиктивный столбец во второй оператор SELECT, чтобы его результат состоял из такого же количества столбцов того же типа, что и для первого SELECT. Пусть это будут значения NULL, хотя с тем же успехом можно было использовать и 0 (ноль). Позже будет показано, почему сделан именно такой выбор.

Вот завершенный запрос:

```
SELECT i.item_id, s.quantity FROM item i, stock s WHERE i.item_id = s.item_id
UNION
SELECT i.item_id, NULL FROM item i WHERE i.item_id NOT IN
    (SELECT i.item_id FROM item i, stock s WHERE i.item_id = s.item_id);
```

Выглядит сложно, но попробуем выполнить его:

```
bpsimple=# SELECT i.item_id, s.quantity FROM item i, stock s WHERE i.item_id
bpsimple-# = s.item_id
bpsimple-# UNION
bpsimple-# SELECT i.item_id, NULL FROM item i WHERE i.item_id NOT IN (select
bpsimple-# i.item_id FROM item i, stock s WHERE i.item_id = s.item_id);
 item_id | quantity
---------+----------
       1 |       12
       2 |        2
       3 |
       4 |        8
       5 |        3
```

```
       6 |
       7 |         8
       8 |        18
       9 |
      10 |         1
      11 |
(11 rows)

bpsimple=#
```

На заре развития SQL это был практически единственный способ решения подобных задач, только SQL89 не поддерживал значения NULL, участвующие во втором операторе SELECT. К счастью, большинство производителей разрешают применение NULL, иначе жизнь стала бы еще сложнее. Если бы не разрешались значения NULL, следующим лучшим вариантом был бы 0, но NULL лучше, т. к. 0 является потенциально дезориентирующим, а NULL всегда остается пустым.

Чтобы избежать такого сложного решения для довольно часто встречающихся задач, производители придумали то, что затем назвали внешними объединениями. К сожалению, поскольку они не входят в стандарт, каждый придумал свое собственное решение с похожими идеями, но разным синтаксисом.

В Oracle и DB2 применялась такая структура: знак '+' в инструкции WHERE показывал, что все значения таблицы должны быть выведены, даже если объединение не удалось. В Sybase символы '*=' в инструкции WHERE указывали на использование внешнего объединения. Обе эти конструкции достаточно просты, но, к несчастью, различны, что не очень-то хорошо для переносимости нашего SQL-кода.

В появившемся позже стандарте SQL92 был определен универсальный способ реализации внешних объединений, с еще одной версией синтаксиса. Производители не торопились реализовывать новый стандарт. Он не поддерживался ни в Sybase 11, ни в Oracle 8, хотя оба эти продукта появились после стандарта. PostgreSQL реализует стандартный метод начиная с версии 7.1 и далее. Поэтому тем, кто работает с более ранней версией, придется установить новую, чтобы попробовать выполнить примеры последнего раздела данной главы. Вероятно, стоит установить новую версию, если номер текущей ниже, чем 7.1, т. к. именно в версию 7.1 внесены значительные исправления и усовершенствования.

В синтаксической структуре SQL92 привычную инструкцию WHERE для объединения таблиц заменяет инструкция ON с ключевыми словами LEFT OUTER JOIN.

Синтаксис таков:

```
SELECT columns FROM table1 LEFT OUTER JOIN table2 ON table1.column =
table2.column
```

Таблица, название которой стоит слева от LEFT OUTER JOIN, – это всегда та таблица, строки которой выбираются все без исключения.

Теперь можно переписать наш запрос, руководствуясь новым синтаксисом:

```
SELECT i.item_id, s.quantity FROM item i LEFT OUTER JOIN stock s ON i.item_id =
s.item_id;
```

Не слишком ли просто, чтобы быть правдой? Давайте выполним запрос:

```
bpsimple=# SELECT i.item_id, s.quantity FROM item i LEFT OUTER JOIN stock s
bpsimple-# ON i.item_id = s.item_id;
 item_id | quantity
---------+----------
       1 |       12
       2 |        2
       3 |
       4 |        8
       5 |        3
       6 |
       7 |        8
       8 |       18
       9 |
      10 |        1
      11 |
(11 rows)

bpsimple=#
```

Отлично, ответ совпадает. Можно понять, почему большая часть производителей решили, что необходимо реализовать внешнее объединение, даже несмотря на то, что его не было в стандарте SQL89.

Существует и эквивалентная конструкция RIGHT OUTER JOIN, но применяется почти всегда левое объединение, вероятно, потому, что, по крайне мере для представителей Запада, более логичным представляется перечисление известных товаров в левой, а не в правой части вывода.

Попробуйте сами. Более сложное условие

Замечательно, что можно использовать такое левое внешнее объединение, как было показано выше, но как добавить более сложные условия?

Предположим, что нам нужны только те строки таблицы stock, которые относятся к товарам, присутствующим на складе более чем в 2-х экземплярах, кроме того, мы будем выбирать только те строки, в которых значение себестоимости превышает 5,0. Это сложная задача, поскольку надо применить к таблице item одно правило (себестоимость > 5.0),

а к таблице stock – другое (количество > 2), и при этом мы все еще хотим составить список всех строк таблицы item, удовлетворяющих некоторому условию, даже если соответствующих им товаров вообще нет на складе.

Скомбинируем условия ON, которые работают для таблиц, связанных левым внешним объединением, с условиями WHERE, ограничивающими все возвращенные строки после выполнения объединения таблиц.

Условие для таблицы stock представляет собой часть внешнего объединения, а накладывать ограничение на строки, в которых нет количества, не надо, поэтому запишем его как часть условия ON:

```
ON i.item_id = s.item_id AND s.quantity > 2
```

Условие для item относится ко всем строкам, используем для него инструкцию WHERE:

```
WHERE i.cost_price > CAST(5.0 AS NUMERIC(7,2));
```

Объединив их вместе, получим:

```
bpsimple=# SELECT i.item_id, i.cost_price, s.quantity FROM item i LEFT OUTER
bpsimple-# JOIN stock s ON i.item_id = s.item_id AND s.quantity > 2 WHERE
bpsimple-# i.cost_price > CAST(5.0 AS NUMERIC(7,2));
 item_id | cost_price | quantity
---------+------------+----------
       1 |      15.23 |       12
       2 |       7.45 |
       5 |       7.54 |        3
       6 |       9.23 |
       7 |      13.36 |        8
      11 |      19.73 |
(6 rows)

bpsimple=#
```

Как это работает

Использовано внешнее объединение LEFT OUTER JOIN для получения всех величин таблицы item, объединенной с таблицей stock для строк, присутствующих в ней и имеющих значение столбца quantity, превышающее 2. Получен набор строк, в котором присутствуют все строки таблицы item, при этом в столбце quantity (из таблицы stock) будет содержаться значение NULL, если только в нем нет записи для этого товара и его количество больше 2. Затем применяется условие WHERE, пропускающее только те строки, в которых себестоимость (из таблицы item) превышает 5.0.

Резюме

В начале главы были описаны агрегатные функции, предназначенные в SQL для выбора отдельных значений из множества строк. В частности, рассматривалась функция `COUNT(*)`, широко применяемая для определения количества строк таблицы.

Затем была введена инструкция `GROUP BY`, позволяющая выбирать группы строк для применения агрегатной функции, и инструкция `HAVING`, накладывающая ограничение на вывод строк, содержащих некоторые агрегированные значения. Потом было вкратце представлено объединение `UNION`, посредством которого можно объединить вывод двух запросов в одно результирующее множество. Оно применяется не часто, но в некоторых случаях может быть очень полезно.

Обсуждались подзапросы, где результаты одного запроса используются в другом. Мы рассмотрели несколько простых примеров; также был затронут гораздо более сложный вид запросов – связанный запрос, в котором один столбец присутствует в обеих частях подзапроса.

В конце мы познакомились с внешними объединениями – очень важной функциональностью, позволяющей осуществлять объединения таблиц, извлекая строки из первой таблицы, даже если объединение со второй не удается.

Эта глава охватывает некоторые сложные аспекты SQL. Не волнуйтесь, если что-то еще не до конца вам понятно. Один из лучших способов по-настоящему понять SQL – это использовать его, причем как можно больше. Инсталлируйте PostgreSQL, установите тестовую базу данных, загрузите данные и принимайтесь за эксперименты.

Об операторе `SELECT` рассказано настолько подробно, насколько необходимо для данной книги. Несмотря на то что это книга для начинающих, представлен широкий диапазон синтаксических структур SQL, поэтому, встретив какую-нибудь сложную конструкцию SQL в реальных системах, вы сможете по крайней мере понять, из чего она состоит.

В главе 8 мы более подробно поговорим о типах данных, о создании таблиц и о том, что необходимо знать для построения собственной базы данных.

8

Определение данных и манипулирование ими

До сих пор наше внимание было сконцентрировано на инструментарии PostgreSQL и манипулировании данными. Несмотря на то что база данных была создана в самом начале этой книги, создание таблиц и типы данных, доступные в PostgreSQL, были рассмотрены поверхностно. Для определения таблиц применялись первичные ключи, некоторые столбцы были определены, как не допускающие использования `Null`-значений.

В базе данных одним из приоритетов должно быть качество данных, поэтому здесь будет более подробно рассказано об имеющихся в PostgreSQL типах данных и о том, как с ними обращаться, в том числе, как осуществлять **распределение** (casting) по типам.

Необходимо на более формальном уровне рассмотреть управление таблицами и использование таких дополнительных возможностей, как **ограничения**, которые позволяют значительно сузить диапазон допустимых операций, применяемых к добавляемым или удаляемым из таблиц данным.

Наличие очень строгих правил, которым подчиняются данные на самом низком уровне, является одной из наиболее эффективных мер по поддержанию непротиворечивости данных. Кроме того, это свойство отличает настоящую базу данных от индексированных файлов, электронных таблиц и т. д.

В главе будут описаны:

- Типы данных
- Специальные типы PostgreSQL

- Управление таблицами
- Представления
- Ограничения на внешние ключи

Типы данных

PostgreSQL поддерживает стандартный набор типов данных SQL, а также некоторые более эзотерические типы, которые мы упомянем, но подробно рассказывать о них не будем, т. к. область их применения достаточно ограниченна.

На самом элементарном уровне PostgreSQL поддерживает следующие типы данных:

- **Логический**
- **Символьный**
- **Числовой**
- **Даты и времени**
- **Специальные типы PostgreSQL**
- **Binary Large Object (BLOB)**

Последовательно рассмотрим каждый из них, кроме Binary Large Objects, поскольку описание этого типа выходит за рамки книги начального уровня.

Логический тип

Логический тип, вероятно, является самым простым типом данных. Он может хранить значения `True` (истина), `False` (ложь) и нашего старого знакомого, `Null`, для неизвестных.

Объявление типа для логического столбца выглядит так: `bool`.

При вставке данных в столбец логического типа PostgreSQL достаточно гибко подходит к тому, что интерпретировать как истину и ложь. Как и в зарезервированных словах SQL, регистр клавиатуры не учитывается (табл. 8.1):

Таблица 8.1. Логическая интерпретация значений

Интерпретируется как True	Интерпретируется как False
`TRUE`	`FALSE`
`'1'`	`'0'`
`'yes'`	`'no'`
`'y'`	`'n'`
`'true'`	`'false'`
`'t'`	`'f'`

Все остальное (кроме NULL) будет отклонено.

При отображении содержимого логического столбца PostgreSQL выводит только t, f, и NULL для истины, лжи и неизвестной величины соответственно. Вне зависимости от того, как именно заданы значения столбца, PostgreSQL хранит одно из трех возможных состояний: True, False или Null, то есть конкретное выражение (true, y, t и т. д.), которое было введено, не запоминается, а хранится только итоговое значение. Это не вызывает никаких трудностей, т. к. столбец объявлен принадлежащим к логическому типу, то нет смысла заботиться о том, какое значение было указано для него изначально.

Попробуйте сами. Логические значения

Создадим простую таблицу со столбцом типа bool и поэкспериментируем со значениями. Вы уже знаете, что для создания столбца, имеющего название и тип, надо просто указать имя и, через пробел, тип.

В таблице testtype предусмотрим такие столбцы: строка переменной длины и логический. Вставим в нее данные и посмотрим, что сохранит PostgreSQL. Вместо того чтобы проводить опыты на базе bpsimple, содержащей реальные данные, создадим базу данных test, которая будет использоваться как место для экспериментов. Те, кто прорабатывал примеры из главы 3, эту базу данных, вероятно, уже создали. Если же нет, создать ее не составит труда:

```
bpsimple=# CREATE DATABASE test;
CREATE DATABASE

bpsimple=# \c test
You are now connected to database test.

test=#
```

Вот последовательность наших действий:

```
test=# CREATE TABLE testtype (
test-# valused VARCHAR(10),
test-# boolres BOOL
test-# );
CREATE
test=# INSERT INTO testtype VALUES('TRUE', TRUE);
INSERT 19132 1
test=# INSERT INTO testtype VALUES('1', '1');
INSERT 19133 1
test=# INSERT INTO testtype VALUES('t', 't');
INSERT 19134 1
test=# INSERT INTO testtype VALUES('no', 'no');
INSERT 19135 1
test=# INSERT INTO testtype VALUES('f', 'f');
INSERT 19136 1
```

```
test=# INSERT INTO testtype VALUES('Null', NULL);
INSERT 19137 1
test=# INSERT INTO testtype VALUES('FALSE', FALSE);
INSERT 19138 1
test=#
```

Проверим, вставлены ли данные:

```
test=# SELECT * FROM testtype;
 valused | boolres
---------+---------
 TRUE    | t
 1       | t
 t       | t
 no      | f
 f       | f
 Null    |
 FALSE   | f
(7 rows)

test=#
```

Как это работает

Создана таблица `testtype`, состоящая из двух столбцов, один из которых хранит строку, а второй – логическое значение. В таблицу вставляются данные, при этом каждый раз первое значение вводится как строковое (в качестве напоминания о том, что именно вводится), вторым же вводится такое же значение, но оно должно быть сохранено как логическое. Также вставлено значение `NULL`, чтобы показать, что PostgreSQL (в отличие от как минимум одной коммерческой базы данных) разрешает хранение величин `NULL` в столбце логического типа. Затем данные извлекаются и становится очевидным, что сохраненные в PostgreSQL значения были интерпретированы как `TRUE`, `FALSE` или `NULL`.

Строковый тип

В любой базе данных чаще всего, наверное, используется именно строковый тип данных, который разделяется на три подтипа:

- отдельные символы
- строки фиксированной длины
- строки переменной длины

Это стандартные строковые типы SQL. Кроме того, PostgreSQL поддерживает еще тип `TEXT`, который похож на тип переменной длины, только для него не надо объявлять верхний предел длины. Тип `TEXT` не является стандартным в SQL. Используйте его с осторожностью. Стандартные типы определяются как `CHAR`, `CHAR(N)` и `VARCHAR(N)`(табл. 8.2).

Таблица 8.2. Символьные типы данных

Определение	Значение
CHAR	Отдельный символ
CHAR(N)	Строка длиной ровно N символов, дополняется пробелами. При попытке сохранить слишком длинную строку лишние символы будут проигнорированы.
VARCHAR(N)	Строка символов длиной менее или равной N, пробелами не дополняется.
TEXT	В действительности символьная строка неограниченной длины, как и VARCHAR, только не надо указывать максимальную длину.

Практически максимальная длина строки для хранения не ограничена в версии 7.1 и выше. Фактически начиная с PostgreSQL версии 7.1 предельный размер был увеличен до 1 Гбайт для любого отдельного поля таблицы, но в реальной жизни вам никогда не понадобится хранить столь длинную строку символов.

Если имеется возможность выбрать один из трех стандартных типов для символьной строки, то какому из них отдать предпочтение? Как обычно, однозначного ответа на этот вопрос не существует.

Если вы уверены в том, что база данных всегда будет работать в PostgreSQL, то можно использовать тип TEXT, он прост в применении и не требует размышлений о максимальной длине. Его длина ограничена лишь максимальным размером строки, поддерживаемым PostgreSQL. Если версия младше 7.1, то предельный размер строки равен приблизительно 8 Кбайт, если только не было указано другое значение перед компиляцией. Начиная с версии 7.1 это ограничение отменено. Еще один хороший повод для установки новой версии! Но есть и обратная сторона медали – TEXT не является стандартным типом, поэтому если есть вероятность, что однажды база данных будет переноситься в другую СУБД, то лучше избегать применения TEXT. В данной книге тип TEXT не встречается, предпочтение отдано более стандартным типам SQL.

Преимущество типа VARCHAR(N) в том, что хранится столько символов, сколько нужно, плюс длина строки. Если же применяется тип CHAR(N), то длина фиксированна и в некоторых случаях это приводит к небольшому увеличению производительности. В общем, если строка достаточно короткая и длина ее известна, то, вероятно, лучше остановить выбор на типе CHAR(N). Если же длина символьной строки существенно меняется от одной строки таблицы к другой, лучше предпочесть тип VARCHAR(N). Испытывая сомнения, используйте VARCHAR(N).

Как и логический тип, все символьные типы допускают хранение NULL для неизвестных величин.

Попробуйте сами. Строковые типы

Для начала удалим таблицу `testtype`, чтобы пересоздать ее со столбцами других типов:

```
test=# DROP TABLE testtype;
DROP
test=# CREATE TABLE testtype (
test-#      singlechar      CHAR,
test-#      fixedchar       CHAR(13),
test-#      variablechar    VARCHAR(128)
test-# );
CREATE
test=# INSERT INTO testtype VALUES('F', '0-349-10177-9', 'The Wasp
test-# Factory');
INSERT 19164 1
test=# INSERT INTO testtype VALUES('S', '1-85723-457-X', 'Excession');
INSERT 19165 1
test=# INSERT INTO testtype VALUES('F', '0-349-10768-8', 'Whit');
INSERT 19166 1
test=# INSERT INTO testtype VALUES(NULL, '', 'T.B.D.');
INSERT 19167 1
test=# SELECT * FROM testtype;

 singlechar |   fixedchar   |   variablechar
------------+---------------+------------------
 F          | 0-349-10177-9 | The Wasp Factory
 S          | 1-85723-457-X | Excession
 F          | 0-349-10768-8 | Whit
            |               | T.B.D.
(4 rows)
test=#
```

Как это работает

Создана таблица, состоящая из трех столбцов, все они принадлежат различным стандартным типам SQL. Столбец `singlechar` хранит отдельный символ, `fixedchar` хранит ровно 13 символов, а `variablechar` может хранить до 128 символов. В столбцы вставлены данные, затем они извлечены, для того чтобы продемонстрировать, что PostgreSQL корректно сохранила их, хотя в действительности в выводе `psql` дополнения пробелами до фиксированной длины не видны.

Числовой тип

Числовые типы PostgreSQL несколько сложнее, чем типы, рассмотренные ранее, но все же достаточно понятны. Существует два отличных друг от друга типа чисел, которые могут храниться в базе данных: **целые** числа и числа **с плавающей точкой**. Целые числа, в свою очередь, подразделяются на специальный подтип `SERIAL`, который уже

применялся для создания уникальных значений в таблице, и целые числа различной длины. Числа с плавающей точкой также разделяются на обычные значения с плавающей точкой и числа с заданной точностью, включая особый тип PostgreSQL – MONEY.

Чтобы охватить всю широту выбора, представим все числовые типы в табл. 8.3:

Таблица 8.3. Числовые типы данных

Тип	Подтип	Стандартное название	Описание
Целые числа	small integer	SMALLINT	Двухбайтное целое со знаком, может хранить числа от -32768 до 32767.
	integer	INT	Четырехбайтное целое, может хранить числа от -2147483648 до +2147483647.
	serial	SERIAL	Как и INT, за исключением того, что обычно он автоматически вводится PostgreSQL (см. главу 6).
Числа с плавающей точкой	float	FLOAT(N)	Число с плавающей точкой с минимальной точностью N, занимающее не более 8 байт памяти.
	float8	REAL	Число с плавающей точкой удвоенной точности (8 байт).
	numeric	NUMERIC(P,S)	Вещественное число из P разрядов, при этом S из них – после десятичной точки. В отличие от FLOAT, это всегда число с заданной точностью, но работа с ним менее эффективна, чем с обычными числами с плавающей точкой.
	money	NUMERIC(9,2)	Специальный тип PostgreSQL, хотя он распространен и в других СУБД. В PostgreSQL MONEY – это альтернативное название для NUMERIC(9,2).

Разделение на целые числа и числа с плавающей точкой вполне понятно, а вот предназначение типа NUMERIC менее очевидно.

Числа с плавающей точкой хранятся в экспоненциальном представлении в виде **мантиссы** и **порядка**. Для типа NUMERIC указываются точность и точное количество разрядов, которое необходимо хранить при проведении расчетов. Можно указать и количество хранимых разрядов после запятой. Реальное местоположение десятичной точки теперь может быть любым!

Существует распространенное заблуждение, согласно которому NUMERIC(5,2) ***может хранить такое число, как 12345,12. Это неверно. Общее количество разрядов равно пяти, так что объявление*** NUMERIC(5,2) ***может хранить значение не более чем 999,99 (до переполнения).***

PostgreSQL в большинстве случаев обнаруживает попытки ввести величины в поля, которые не могут их сохранить, поэтому ввести слишком большое число в любой числовой столбец не удастся.

Попробуйте сами. Числовые типы

Сначала удалим таблицу `testtype`, чтобы затем пересоздать ее со столбцами других типов:

```
test=# DROP TABLE testtype;
DROP
test=# CREATE TABLE testtype (
test-#      asmallint    SMALLINT,
test-#      anint        INT,
test-#      afloat       FLOAT(2),
test-#      areal        REAL,
test-#      anumeric     NUMERIC(5,2)
test-# );
CREATE
test=# INSERT INTO testtype VALUES(2, 2, 2.0, 2.0, 2.0);
INSERT 19244 1
test=# INSERT INTO testtype VALUES(-100, -100, 123.456789, 123.456789,
test-# 123.456789);
INSERT 19245 1
test=# INSERT INTO testtype VALUES(-32768, -123456789, 1.23456789,
test-# 1.23456789, 1.23456789);
INSERT 19246 1
test=# INSERT INTO testtype VALUES(-32768, -123456789, 123456789.123456789,
test-# 123456789.123456789, 123456789.123456789);
ERROR:  overflow on numeric ABS(value) >= 10^8 for field with precision 5
scale 2
test=# INSERT INTO testtype VALUES(-32768, -123456789, 123456789.123456789,
test-# 123456789.123456789, 123.123456789);
INSERT 19247 1
test=# SELECT * FROM testtype;

 asmallint |   anint    |   afloat    |    areal    | anumeric
-----------+------------+-------------+-------------+-----------
         2 |          2 |           2 |           2 |      2.00
      -100 |       -100 |     123.457 |     123.457 |    123.46
    -32768 | -123456789 |     1.23457 |     1.23457 |     1.23
    -32768 | -123456789 | 1.23457e+08 | 1.23457e+08 |    123.12
(4 rows)

test=#
```

Как это работает

Создана таблица со столбцами следующих типов: короткое целое, обычное целое, число с плавающей точкой, вещественное число и пятизначное число с двумя значащими цифрами после запятой.

Видно, что FLOAT и REAL ведут себя одинаково, а вот поведение столбца типа NUMERIC немного отличается. Вместо того чтобы хранить приблизительные значения, он округляет число и сохраняет определенное количество знаков после десятичной точки. Если попытаться вставить слишком большое число, то INSERT не будет выполнен. Обратите внимание, что как FLOAT, так и REAL округляют числа, например 123.456789 было округлено до 123.457.

Типы даты и времени

Типы даты и времени (типы, хранящие информацию, связанную с временем) уже встречались нам при рассмотрении форматов времени.

В PostgreSQL существует целый ряд типов, имеющих отношение к дате и времени, но здесь ограничимся стандартными типами SQL-92 (табл. 8.4):

Таблица 8.4. Типы даты и времени

Определение	Значение
DATE	Хранит сведения о дате
TIME	Хранит информацию о времени
TIMESTAMP	Хранит дату и время
INTERVAL	Хранит информацию о временном интервале

О дате и времени достаточно подробно рассказано в главе 4, поэтому не будем останавливаться на них сейчас.

Специальные типы PostgreSQL

Будучи по своей природе исследовательской СУБД, PostgreSQL имеет ряд нестандартных типов данных. Перечислим просто их ради интереса (табл. 8.5), но не будем вдаваться в детали. За подробной информацией обратитесь к руководству пользователя по PostgreSQL, доступному в оперативной справке:

Таблица 8.5. Специальные типы PostgreSQL

Определение	Значение
Box	Прямоугольник
cidr или inet	Интернет-адрес Ipv4, такой как 196.192.12.45
Line	Набор точек
Point	Пара чисел, задающая координаты точки
Lseg	Линейный сегмент
polygon	Замкнутая геометрическая линия

Создание собственных типов

PostgreSQL также позволяет создавать собственные типы, применяемые в базе данных, при помощи команды SQL CREATE TYPE. Это не та возможность, к которой обращаются часто, и реализована она уникальным способом, присущим только PostgreSQL. Дополнительную информацию можно найти в документации для пользователей. Знайте, что создание собственных типов приводит к тому, что схема базы данных становится специфичной для PostgreSQL, т. к. типы, созданные пользователем, часто бывают непереносимыми.

Массивы

PostgreSQL имеет еще одно необычное (не входящее в стандартные свойства SQL) свойство – способность хранить в таблицах массивы. Как правило, массивы реализуются при помощи дополнительной таблицы, но поскольку порой они бывают весьма полезны, а работать с ними просто, представим краткое описание массивов PostgreSQL.

Чтобы объявить столбец как массив, просто добавьте квадратные скобки [] после названия типа, указывать количество элементов не обязательно. Предположим, надо завести таблицу служащих с индикатором, показывающим, в какие дни недели они работали. Обычно создается столбец для каждого дня или отдельная таблица для хранения рабочих дней. В PostgreSQL можно упростить ситуацию и хранить непосредственно массив рабочих дней:

```
test=# CREATE TABLE empworkday (
test-# refcode CHAR(5),
test-# workdays INT[]
test-# );
CREATE

test=#
```

Создана таблица empworkday, состоящая из двух столбцов. В одном хранится символьный код ссылки, а в другом – массив целых чисел workdays. Чтобы ввести значения в столбец массива, необходимо заключить список значений, разделенных запятыми, в пару разделителей {}:

```
test=# INSERT INTO empworkday VALUES('val01', '{0,1,0,1,1,1,1}');
INSERT 19290 1
test=# INSERT INTO empworkday VALUES('val02', '{0,1,1,1,1,0,1}');
INSERT 19291 1

test=#
```

Теперь можно выбрать все значения элементов массива за одну попытку:

```
test=# SELECT * FROM empworkday;
 refcode |    workdays
---------+-----------------
```

```
 val01   | {0,1,0,1,1,1,1}
 val02   | {0,1,1,1,1,0,1}
(2 rows)

test=#
```

Или же выбрать отдельные элементы, указав индекс массива:

```
test=# SELECT workdays[2] FROM empworkday WHERE refcode = 'val02';
 workdays
----------
        1
(1 row)

test=#
```

Номер первого элемента массива равен 1 (что необычно). При попытке выбрать несуществующий элемент будет возвращено значение `NULL`.

В руководстве пользователя на веб-сайте можно найти более подробную информацию об уникальной обработке массивов, поддерживаемой PostgreSQL.

Преобразование типов

Время от времени в базе данных возникает необходимость преобразования одного типа в другой. В целом к преобразованиям типов следует относиться с осторожностью, поскольку многочисленные преобразования могут свидетельствовать о недостатках проекта. Однако в некоторых случаях (как мы это видели в главе 4) преобразования необходимы, в частности, для дат.

Реляционные СУБД осуществляют эту операцию различными способами. PostgreSQL использует запись `CAST` и альтернативный синтаксис с двойным двоеточием. Формат таков:

```
CAST(column-name AS type-definition-to-convert-to)
```

или

```
column-name::type-definition-to-convert-to
```

Используется вместо простого имени столбца в операторе `SELECT`.

Предположим, что нужно извлечь данные из таблицы `orderinfo` в виде `char(10)`. Вернемся к базе данных `bpsimple` и просто напишем:

```
SELECT CAST(date_placed AS CHAR(10)) FROM orderinfo;
```

Выполнив этот оператор в базе данных `bpsimple`, увидим:

```
bpsimple=# SELECT CAST(date_placed AS CHAR(10)) FROM orderinfo;
  ?column?
-------------
 2000-03-13
```

```
 2000-06-23
 2000-09-02
 2000-09-03
 2000-07-21
(5 rows)

bpsimple=#
```

Заметьте, что теперь у столбца нет названия.

Можно использовать CAST (или ::) как для столбцов, так и для значений, можно также дать результату операции название, указав заголовок столбца.

Попробуйте сами. Преобразование типов

Предположим, что требуется вывести список товаров с указанием цены, округленной до ближайшего целого доллара, дороже 5 долларов. Можно попробовать «в лоб»:

```
bpsimple=# SELECT sell_price::int FROM item WHERE sell_price > 5.0;
```

PostgreSQL выражает недовольство:

```
ERROR:  Unable to identify an operator '>' for types 'numeric' and 'float8'
        You will have to retype this query using an explicit cast
```

Необходимо преобразовать не только sell_price в тип целых чисел, но и величину, задаваемую для проверки цены, в тип numeric, т. к. это тип столбца sell_price, который проверяется. К тому же не указано имя для результирующего столбца. Попробуем еще раз!

```
bpsimple=# SELECT item_id, sell_price::int AS "Guide Price" FROM item WHERE
bpsimple-# sell_price > 5.0::NUMERIC(7,2);
 item_id | Guide Price
---------+-------------
       1 |          22
       2 |          11
       5 |          10
       6 |          16
       7 |          20
      11 |          25
(6 rows)

bpsimple=#
```

Получилось!

Как это работает

Столбец sell_price приведен к целому типу (sell_price::int) и получил название (AS "Guide Price"). Получился поименованный столбец вывода в формате целых чисел. Чтобы добиться совместимости типов в час-

ти оператора WHERE, мы также применили CAST для преобразования величины 5.0 в NUMERIC(7,2), тип столбца sell_price.

То же самое можно было сделать с помощью записи CAST(::); эти два варианта взаимозаменяемы. Обратите внимание, что универсального способа преобразования типов не существует. Например, дату нельзя привести к типу целых чисел.

Другие операции с данными

В PostgreSQL также есть несколько стандартных функций, пригодных для обработки столбцов. Не будем приводить здесь полный перечень, подробную информацию можно найти в документах, доступных в оперативной справке.

Представим только те, которые оказываются полезными чаще всего (табл. 8.6):

Таблица 8.6. Стандартные функции обработки данных

Функция	Описание
length(column-name)	Возвращает длину символьной строки.
trim(column-name)	Удаляет начальный и замыкающий пробелы.
strpos(column-name, string)	Возвращает позицию символьной строки в столбце.
substr(column-name, position, length)	Возвращает length символов строки, начиная поиск с символа, позиция которого задана. Первый символ соответствует позиции 1.
round(column-name, length)	Округляет число до указанного количества десятичных разрядов.
Abs(number)	Возвращает абсолютное значение числа.

Они используются так же, как и функция CAST:

```
bpsimple=# SELECT substr(description, 3, 5), round(sell_price, 1) FROM item;
 substr | round
--------+-------
 od Pu  |  22.0
 bik C  |  11.5
 nux C  |   2.5
 ssues  |   4.0
 cture  |  10.0
 n Sma  |  15.8
 n Lar  |  20.0
 othbr  |   1.5
 man C  |   2.5
 rrier  |   0.0
 eaker  |  25.3
(11 rows)

bpsimple=#
```

Магические переменные

Время от времени требуется сохранить в базе данных информацию, имеющую некоторое отношение к текущему пользователю или времени, в частности для целей аудита.

PostgreSQL для такого случая имеет в запасе четыре магические переменные:

- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- CURRENT_USER

Их можно использовать аналогично названиям столбцов, а можно выбирать их (посредством SELECT), вообще не указывая название таблицы:

```
bpsimple=# SELECT item_id, quantity, CURRENT_TIMESTAMP FROM stock;
 item_id | quantity |       timestamp
---------+----------+------------------------
       1 |       12 | 2001-03-25 09:22:11+01
       2 |        2 | 2001-03-25 09:22:11+01
       4 |        8 | 2001-03-25 09:22:11+01
       5 |        3 | 2001-03-25 09:22:11+01
       7 |        8 | 2001-03-25 09:22:11+01
       8 |       18 | 2001-03-25 09:22:11+01
      10 |        1 | 2001-03-25 09:22:11+01
(7 rows)

bpsimple=# SELECT CURRENT_USER, CURRENT_TIME;
 current_user |   time
--------------+----------
 rick         | 09:22:20
(1 row)

bpsimple=#
```

Магические переменные также можно использовать в операторах INSERT и UPDATE:

```
INSERT INTO orderinfo(orderinfo_id, customer_id, date_placed, date_shipped,
shipping) VALUES (5, 8, CURRENT_DATE, NULL, 0.0);
```

Столбец OID

Вы должны были заметить, что каждый раз в ответ на вставку данных PostgreSQL выводит выглядящее почти произвольным образом число (а также количество вставленных строк). Это внутренний номер, который PostgreSQL хранит для каждой записи, этот обычно невидимый столбец называется oid.

Во многих реляционных базах данных или нет такого столбца, или же он недоступен пользователям. Что касается PostgreSQL, то можно

увидеть эти номера, явно указав название столбца при выполнении выборки из таблицы, например:

```
bpsimple=# SELECT oid, fname, lname FROM customer;
  oid  |   fname   |  lname
-------+-----------+--------
 19888 | Jenny     | Stones
 19889 | Andrew    | Stones
 19890 | Alex      | Matthew
 19891 | Adrian    | Matthew
 19892 | Simon     | Cozens
 19893 | Neil      | Matthew
 19894 | Richard   | Stones
 19895 | Ann       | Stones
 19896 | Christine | Hickman
 19897 | Mike      | Howard
 19898 | Dave      | Jones
 19899 | Richard   | Neill
 19900 | Laura     | Hendy
 19901 | Bill      | O'Neill
 19902 | David     | Hudson
(15 rows)

bpsimple=#
```

В другой базе данных значения столбца `oid` почти наверняка будут другими. OID также можно встретить в конфигурации ODBC-драйвера. Можно выбирать, показывать его или сделать невидимым.

В правильно спроектированной базе данных с разумно созданными первичными ключами применять OID никогда не понадобится. Этот идентификатор упомянут для полноты картины, настоятельно советуем вам удержаться от искушения использовать его когда бы то ни было.

Манипулирование таблицами

Познакомившись с типами данных PostgreSQL, используем их для создания таблиц. Вам уже встречалась SQL-команда `CREATE TABLE` (с ее помощью создавались таблицы в учебной базе данных), сейчас поговорим о ней на более формальном уровне. Кроме того, рассмотрим такие дополнительные возможности, как временные таблицы, изменение таблиц после создания и, конечно же, удаление таблиц, ставших ненужными.

Создание таблицы

Базовый синтаксис оператора, создающего таблицы, таков:

```
CREATE [TEMPORARY] TABLE table-name (
     { column-name type [ column-constraint ] [,…] }
     [ CONSTRAINT table-constraint ]
) [ INHERITS (existing-table-name) ]
```

Запись хотя и короткая, но выглядит достаточно сложно, на самом же деле все просто и понятно. В первой строке просто сообщается, что мы создаем таблицу посредством CREATE TABLE, указано название таблицы, затем идет открывающая скобка. К TEMPORARY вернемся несколько позже. После этого перечисляются название столбца, его тип и необязательное ограничение для столбца. По существу, в таблице может быть сколько угодно столбцов, каждый из них должен быть отделен от других запятыми. Необязательное ограничение для столбца позволяет указать дополнительные правила, налагаемые на него, самый распространенный пример – NOT NULL – нам уже встречался.

За перечнем столбцов следует необязательное ограничение для таблицы, позволяющее записать дополнительные правила уровня таблиц, которым должны подчиняться данные, содержащиеся в таблице.

Последним идет расширение PostgreSQL, INHERITS, благодаря которому новая таблица может быть создана на основе уже имеющейся. Создаваемая таблица наследует столбцы от уже существующих таблиц. Новая таблица содержит вдобавок к явно указанным столбцам все столбцы, имеющиеся в таблицах, перечисленных после ключевого слова INHERITS. Более полные сведения об INHERITS можно найти в документах из оперативной справки.

Настоятельно рекомендуем сохранять команды, применяемые для создания базы данных, в файлах сценариев и всегда создавать базы данных при помощи этих сценариев. Если понадобится изменить проект базы данных, то гораздо проще и безопаснее изменить файл сценария, а затем пересоздать базу, чем проверять и заново вызывать команды, которые выполнялись несколько месяцев (или дней) тому назад. Усилия, затраченные на создание файла сценария и его обновление, окупятся сторицей.

Ограничения для столбцов

В этой главе уже было представлено множество основных команд создания таблиц, поэтому теперь перейдем прямо к рассмотрению наиболее часто встречающихся ограничений на столбцы, которые могут вам понадобиться. Какие-то столбцы таблицы нередко подчиняются определенным правилам, и это нормально. Некоторые простые правила нам уже встречались, например для столбца с фамилией клиента гарантировалось, что хранимые в нем величины не равны NULL. В некоторых случаях правила накладываются и на известные данные. Например для того, чтобы заставить столбец со ставкой заработной платы принимать только значения, превышающие некоторую минимальную величину, или же для обеспечения уникальности столбцов. Применение ограничений (constraints) к столбцам позволяет проводить подобные проверки на самом нижнем уровне приложения – в базе данных. Введение быстрых и строгих основных правил на уровне базы данных хорошо тем, что этот уровень не зависит от приложения, поэтому лю-

бые ошибки приложения, которые могли бы вызвать появление недопустимых величин, будут «выловлены» базой данных. К тому же, часто бывает проще написать определение при создании таблицы, чем программировать поддержку правила в приложении.

Приведем основные ограничения, которые могут вам пригодиться (табл. 8.7). Надо сказать, что существуют и более сложные ограничения, о которых можно прочитать в документации из оперативной справки:

Таблица 8.7. Основные ограничения для столбцов

Определение	Значение
`NOT NULL`	В столбце не может быть сохранено значение `NULL`.
`UNIQUE`	Значение, сохраняемое в столбце, должно отличаться от всех остальных строк в базе данных. Об обработке `NULL`-значений см. ниже.
`PRIMARY KEY`	Комбинация `NOT NULL` и `UNIQUE`. В каждой таблице только один столбец может быть помечен как `PRIMARY KEY` (а вот столбцов, помеченных одновременно как `NOT NULL` и `UNQIUE`, может быть несколько). Дальше в этой главе будет рассказано о том, что при необходимости создать составной первичный ключ (первичный ключ, заключающий в себе несколько столбцов) следует использовать ограничение уровня таблицы, а не ограничения для столбца, обсуждаемые в настоящий момент.
`DEFAULT value`	Позволяет задавать значение по умолчанию при вводе данных.
`CHECK (condition)`	Позволяет осуществлять проверку условия при вводе или обновлении данных.
`REFERENCES`	См. раздел «Ограничения – внешние ключи» далее в этой главе.

Все ограничения, кроме `REFERENCES`, о котором будет подробно рассказано чуть позже, достаточно просты и понятны. В таблице может быть сколько угодно столбцов с любым количеством ограничений (за исключением `PRIMARY KEY`). Возможно именовать ограничения столбцов, но эта возможность используется редко.

Обратим внимание на одну особенность, имеющую место при добавлении значения `NULL` в столбец с ограничением `UNIQUE`. PostgreSQL рассматривает каждое `NULL`-значение как уникальное, поэтому в столбце, объявленном как `UNIQUE`, может быть сколь угодно много строк с `NULL`. Стандарт SQL допускает использование только одного `NULL`-значения, так что это некоторое отклонение от стандарта. Вероятно, позиция стандарта SQL более логична, ведь если величина неизвестна (`NULL`), то нет никакой возможности узнать, равны две такие величины или нет; реализация PostgreSQL же, допускающая несколько `NULL`, может быть более понятна интуитивно.

Попробуйте сами. Ограничения для столбцов

Проще всего понять, что представляют собой ограничения для столбцов, посмотрев на них в действии. Создадим для экспериментов таблицу в созданной ранее базе данных `test` и введем некоторые ограничения:

```
bpsimple=# \c test
You are now connected to database test
test=# CREATE TABLE testcolcons (
test-#        colnotnull INT NOT NULL,
test-#        colunique INT UNIQUE,
test-#        colprikey INT PRIMARY KEY,
test-#        coldefault INT DEFAULT 42,
test-#        colcheck INT CHECK( colcheck < 42)
test-# );
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index
'testcolcons_pkey' for table 'testcolcons'
NOTICE:  CREATE TABLE/UNIQUE will create implicit index
'testcolcons_colunique_key' for table 'testcolcons'
CREATE

test=#
```

Как видите, PostgreSQL сообщает, что созданы индексы для введения в действие ограничений `PRIMARY KEY` и `UNIQUE`.

Итак, таблица с множеством различных ограничений для столбцов создана, и мы попробуем ввести в нее данные, чтобы на практике посмотреть, как работают ограничения:

```
test=# INSERT INTO testcolcons(colnotnull, colunique, colprikey, coldefault,
test-# colcheck) VALUES(1,1,1,1,1);
INSERT 19341 1
test=# INSERT INTO testcolcons(colnotnull, colunique, colprikey, coldefault,
test-# colcheck) VALUES(2,2,2,2,2);
INSERT 19342 1
test=# INSERT INTO testcolcons(colnotnull, colunique, colprikey, coldefault,
test-# colcheck) VALUES(2,2,2,2,2);
ERROR:  Cannot insert a duplicate key into unique index testcolcons_pkey

test=#
```

Последний оператор `INSERT` не был выполнен, т. к. индекс `testcolcons_pkey` обнаружил повторяющееся значение. Применим немного здравого смысла, чтобы осознать, что индекс с названием `testcolcons_pkey` ссылается на первичный ключ таблицы `testcolcons`. Вряд ли от вашей интуиции потребовалось большое усилие! У каждой таблицы может быть только один первичный ключ, поэтому название индекса, образованное как `tablename_pkey`, однозначно.

```
test=# INSERT INTO testcolcons(colnotnull, colunique, colprikey, coldefault,
test-# colcheck) VALUES(2,2,9,2,2);
ERROR:  Cannot insert a duplicate key into unique index
testcolcons_colunique_key

test=#
```

На этот раз успешному исполнению оператора INSERT помешало то, что индекс testcolcons_colunique_key обнаружил повторяющееся значение. Сколь угодно много столбцов может быть объявлено как UNIQUE, поэтому PostgreSQL называет индекс tablename_columnname_key, чтобы было понятно, какой столбец вызывает проблемы:

```
test=# INSERT INTO testcolcons(colnotnull, colunique, colprikey, coldefault,
test-# colcheck) VALUES(2,9,9,2,2);
INSERT 19345 1
test=# INSERT INTO testcolcons(colnotnull, colunique, colprikey, coldefault,
test-# colcheck) VALUES(3,3,3,3,100);
ERROR:  ExecAppend: rejected due to CHECK constraint testcolcons_colcheck

test=#
```

Теперь INSERT не выполнен, потому что не выполнено условие ограничения CHECK. Обратите внимание, что ограничение называется tablename_columnname, поэтому найти источник проблем очень просто:

```
test=# UPDATE testcolcons SET colunique = 1 WHERE colnotnull = 2;
ERROR:  Cannot insert a duplicate key into unique index
testcolcons_colunique_key

test=#
```

Невозможно обновить значение colunique, потому что в таблице уже есть строка, в которой столбец имеет такое значение:

```
test=# INSERT INTO testcolcons(colnotnull, colunique, colprikey, colcheck)
VALUES(3,3,3,41);
INSERT 19346 1
test=# SELECT * FROM testcolcons ;
 colnotnull | colunique | colprikey | coldefault | colcheck
------------+-----------+-----------+------------+----------
          1 |         1 |         1 |          1 |        1
          2 |         2 |         2 |          2 |        2
          2 |         9 |         9 |          2 |        2
          3 |         3 |         3 |         42 |       41
(4 rows)`

test=#
```

И наконец, мы не задали значение для столбца coldefault (заметьте, что его нет в перечне столбцов), и в этом случае подставляется значение по умолчанию.

Для того чтобы проверить ограничения для столбцов таблицы, можно вывести их список при помощи psql, указав команду \d tablename:

```
test=# \d testcolcons
        Table "testcolcons"
 Attribute  |  Type   |  Modifier
------------+---------+------------
 colnotnull | integer | not null
 colunique  | integer |
 colprikey  | integer | not null
 coldefault | integer | default 42
 colcheck   | integer |
Indices: testcolcons_colunique_key,
         testcolcons_pkey
Constraint: (colcheck < 42)

test=#
```

Как это работает

В PostgreSQL есть множество способов реализации ограничений. Проконтролировать порядок проверки ограничений невозможно. Какая именно ошибка будет выведена, зависит от внутренних реализаций PostgreSQL. Можно лишь гарантировать, что все ограничения будут проверены прежде, чем данные будут сохранены в базе. Можно также использовать транзакции (они будут представлены в главе 9 «Транзакции и блокировки»), чтобы убедиться, что в базе данных произведены либо все изменения, либо ни одно.

Ограничения для таблиц

Ограничения для таблиц очень похожи на ограничения для столбцов, но (как понятно из названия) применяются они не к отдельному столбцу, а к таблице в целом. Иногда требуется указать ограничения, такие как первичный ключ, на уровне таблицы, а не столбца. Например, мы уже видели, что в таблице orderline необходимо использовать два столбца, orderinfo_id и item_id, вместе в качестве составного ключа для идентификации строки, т. к. лишь комбинация столбцов уникальна.

Приведем четыре ограничения для таблиц (табл. 8.8):

Таблица 8.8. Ограничения для таблиц

Название	Описание
UNIQUE(column list)	Значение, хранимое в столбцах, должно отличаться от значений всех остальных строк данного столбца.
PRIMARY KEY(column list)	Комбинация NOT NULL и UNIQUE. В каждой таблице может быть только один PRIMARY KEY – либо как ограничение для таблицы, либо как ограничение для столбца.

Таблица 8.8 (продолжение)

Название	Описание
CHECK (condition)	Позволяет осуществлять проверку условия при вводе или обновлении данных.
REFERENCES	См. раздел «Ограничения – внешние ключи» далее в этой главе.

Как видите, это нечто большее чем случайное сходство с ограничениями для столбцов.

Существуют и отличия:

- Ограничения для таблиц перечисляются после всех столбцов
- Ограничения воспринимают списки названий столбцов (разделенных запятыми), поэтому ограничение уровня таблицы может относиться к нескольким столбцам

Теперь давайте займемся ограничениями для таблиц на практике.

Попробуйте сами. Ограничения для таблиц

Сначала создадим таблицу с несколькими ограничениями:

```
test=# CREATE TABLE ttconst (
test-# mykey1 int,
test-# mykey2 int,
test-# mystring varchar(15),
test-# CONSTRAINT cs1 CHECK (mystring <> ''),
test-# CONSTRAINT cs2 PRIMARY KEY(mykey1, mykey2)
test-# );
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'cs2' for table
'ttconst'
CREATE
test=#
```

Обратите внимание, что, как и для ограничения уровня столбца, PostgreSQL создала индекс для введения в действие первичного ключа:

```
test=# INSERT INTO ttconst VALUES(1,1,'Hello');
INSERT 19381 1
test=# INSERT INTO ttconst VALUES(1,2,'Bye');
INSERT 19382 1
test=#
```

Заметьте, что значение `mykey1` равно 1 для обеих строк, но т. к. значение `mykey2` изменилось, то ограничение уникальности пары не нарушено:

```
test=# INSERT INTO ttconst VALUES(1,2,'');
ERROR:  ExecAppend: rejected due to CHECK constraint cs1
test=#
```

Ограничение CHECK для таблицы работает практически идентично аналогичному ограничению для столбца, отклоняя ввод строки из-за наличия пустой символьной строки:

```
test=# INSERT INTO ttconst VALUES(2,2,'Chow');
INSERT 19383 1
test=# INSERT INTO ttconst VALUES(2,2,'Hi');
ERROR:  Cannot insert a duplicate key into unique index cs2

test=#
```

Если совпадают оба значения mykey, строка также отклоняется, т. к. на этот раз нарушено ограничение – первичный ключ.

Как видите, ограничения для таблиц очень похожи на свои эквиваленты для столбцов. В целом, лучше использовать (если этого достаточно) ограничения для столбцов. Если же необходимы ограничения обоих типов (как при создании базы данных bpsimple), то большинство программистов для единообразия предпочитают везде использовать ограничения уровня таблицы.

Изменение структуры таблиц

Жизнь не стоит на месте, поэтому как бы тщательно вы ни выполняли все требования и как бы аккуратно ни реализовывали свою базу данных, наступит день, когда придется изменить структуру таблицы.

Один из способов был описан в главе 6 – можно выполнить оператор INSERT INTO, который собирает данные, выбранные из уже существующей таблицы. Поступим следующим образом:

- Создадим новую рабочую таблицу со структурой, идентичной уже существующей таблице
- Посредством INSERT INTO внесем в рабочую таблицу данные, идентичные данным исходной таблицы
- Удалим существующую таблицу
- Пересоздадим таблицу с тем же именем, внеся необходимые изменения
- Снова применим INSERT INTO, чтобы внести в измененную таблицу данные из рабочей таблицы
- Удалим рабочую таблицу

Очевидно, что работы очень много (особенно если в таблице много данных). Можно сказать, слишком много, даже если все что нужно – это просто добавить в таблицу столбец. Стандарт SQL предоставляет возможность добавления столбцов в уже существующую таблицу и удаления их in situ (на месте). То есть столбец в это время содержит данные. Во время написания книги PostgreSQL поддерживал только добавление столбцов в таблицу, но не их удаление.

Также можно переименовать столбец, сохранив его данные, можно переименовать и всю таблицу.

Синтаксис команды прост:

```
ALTER TABLE table-name ADD COLUMN column-name column-type
ALTER TABLE table-name RENAME COLUMN old-column-name TO new-column-name
ALTER TABLE old-table-name RENAME TO new-table-name
```

Столбцы, добавляемые в таблицу с существующими данными, будут содержать NULL-значения во всех строках.

Посмотрим на команду в действии:

```
test=# \d ttconst
               Table "ttconst"
 Attribute |         Type          | Modifier
-----------+-----------------------+----------
 mykey1    | integer               | not null
 mykey2    | integer               | not null
 mystring  | character varying(15) |
Index: cs2
Constraint: (mystring <> ''::"varchar")

test=#
```

Сначала добавим новый столбец:

```
test=# ALTER TABLE ttconst ADD COLUMN mydate DATE;
ALTER
test=# \d ttconst
               Table "ttconst"
 Attribute |         Type          | Modifier
-----------+-----------------------+-----------
 mykey1    | integer               | not null
 mykey2    | integer               | not null
 mystring  | character varying(15) |
 mydate    | date                  |
Index: cs2
Constraint: (mystring <> ''::"varchar")

test=#
```

Теперь переименуем только что добавленный столбец:

```
test=# ALTER TABLE ttconst RENAME COLUMN mydate TO birthdate;
ALTER
test=# \d ttconst
               Table "ttconst"
 Attribute |         Type          | Modifier
-----------+-----------------------+----------
 mykey1    | integer               | not null
 mykey2    | integer               | not null
 mystring  | character varying(15) |
```

```
 birthdate | date                    |
Index: cs2
Constraint: (mystring <> ''::"varchar")

test=#
```

В заключение переименуем всю таблицу:

```
test=# ALTER TABLE ttconst RENAME TO ttconst2;
ALTER

test=#
```

Как видите, работать с оператором `ALTER TABLE` достаточно просто.

Необходимо очень осторожно относиться к постоянным изменениям табличной структуры за счет введения новых столбцов. Новые столбцы всегда добавляются в конец таблицы, поэтому могут не очень хорошо соответствовать ее логике.

Предположим, что при создании таблицы `customer` был забыт столбец `title` (мистер, миссис и т. д.), его добавили позже. Столбец будет добавлен в конец, из-за этого конструкция таблицы `customer` станет несколько странной, ведь «титул» клиента указан после его номера телефона. Именно по этой причине многие программисты настороженно относятся к добавлению столбцов в существующую таблицу и предпочитают:

- Создать новую таблицу с временным именем, содержащую необходимые столбцы в правильном логическом порядке
- Посредством `INSERT INTO ... SELECT ...` сделать новую таблицу дубликатом изменяемой
- Удалить старую таблицу
- Дать новой таблице имя старой

Учтите, что при удалении и переименовании таблиц также бывает необходимо удалять последовательности и триггеры (см. главу 10 «Хранимые процедуры и триггеры»).

Удаление таблиц

Удалить таблицу чрезвычайно легко:

```
DROP TABLE table-name
```

Престо![1] Таблица вместе со всеми содержащимися в ней данными исчезает. Применяя эту команду, будьте внимательны!

[1] От латинского praestus – готово. – *Примеч. ред.*

Временные таблицы

Все средства SQL, рассмотренные нами до этого момента, организованы так, чтобы обеспечить достижение желаемого результата с помощью единственного, хотя порой и сложного оператора SELECT. Для большинства случаев этот способ весьма неплох. (Вы должны помнить, что SQL – это декларативный язык.) Если определить, каков должен быть результат, SQL найдет наилучший способ для его получения. Но иногда бывает невозможно (или неудобно) сделать все при помощи одного оператора SELECT и приходится хранить некоторые промежуточные результаты.

Обычно в качестве временного хранилища выступает таблица, в которой можно хранить несколько строк. Конечно, всегда можно создать таблицу, выполнить необходимые действия и удалить эту таблицу, но есть опасность, что промежуточная таблица не будет удалена – либо из-за ошибки в приложении, либо просто из-за забывчивости пользователя. В результате появляются «паразитные» таблицы, часто со странными именами, разбросанные по базе данных. К сожалению, не всегда очевидно, какие таблицы предназначены только для промежуточных результатов и, следовательно, могут быть уничтожены, а какие в настоящее время используются.

В SQL предусмотрено простое решение данной проблемы, оно реализуется посредством временных таблиц. Создавайте таблицу при помощи CREATE TEMPORARY TABLE (или CREATE TEMP TABLE, это синонимы), а не CREATE TABLE. Таблица создается так же, как и обычно, за тем лишь исключением, что при завершении сессии и отсоединении от базы данных временная таблица удаляется автоматически.

Знайте, что команда \dt не выводит список временных таблиц.

Представления

Если база данных достаточно сложная или если существует множество пользователей с различными правами доступа (см. главу 11 «Администрирование PostgreSQL»), то может возникнуть необходимость в создании иллюзии таблицы. Рассмотрим пример.

Предположим, что требуется разрешить служащим склада просматривать товары и штрих-коды в базе данных. Сейчас они хранятся в двух таблицах – item и barcode соответственно. С точки зрения проекта базы данных это корректно, но возникает желание предоставить людям данные в более простом виде, например, упрощенно представив их с помощью какого-нибудь пользовательского интерфейса (см. главу 5). Вместо того чтобы изменять схему базы данных, можно просто создать иллюзию одной таблицы, именно это и делается посредством **представления** (view).

Синтаксис команды, создающей представление, очень прост:

```
CREATE VIEW name-of-view AS select-statement;
```

Теперь можно работать с представлением так, как если бы это была таблица. На момент написания книги представления в PostgreSQL были доступны только для чтения. В других же СУБД представления и, следовательно, лежащие в их основе данные могут обновляться, подобно таблицам. Можно выбирать данные (с помощью SELECT) из представления так же, как из таблицы, в том числе, объединяя его с другими таблицами и используя инструкцию WHERE.

При каждой выборке данных с помощью представления данные перестраиваются заново, поэтому они всегда актуальны (то есть это не застывшая копия, сохраненная в момент создания представления).

Предположим, что требуется создать представление, которое реализовывало бы упрощенный вариант таблицы item. Мы хотим, чтобы отображались только item_id, description и sell_price. Оператор SELECT будет выглядеть так:

```
SELECT item_id, description, sell_price FROM item;
```

Для того чтобы создать представление с названием (например) item_price, следует написать:

```
CREATE VIEW item_price AS SELECT item_id, description, sell_price FROM item;
```

Если вы помните, в главе 5 обсуждались небольшие трудности, возникшие с определением цены в таблице item. Будем считать, что для «корректности» цена определена как NUMERIC(7,2). Можно сохранить это определение, но в представлении показывать другой тип (при создании представления используем CAST в операторе SELECT).

Попробуйте сами. Создание представления

Создадим представление таблицы item, в котором были бы изменены два аспекта в картине, которую видит пользователь. Во-первых, скроем столбец cost_price (права доступа к исходной таблице item и способы его закрытия для обычных пользователей подробно описаны в главе 11 «Администрирование PostgreSQL»). Во-вторых, представим sell_price как просто число с плавающей точкой, а не как тип NUMERIC.

Создадим представление следующим образом:

```
bpsimple=# CREATE VIEW item_price AS SELECT item_id, description,
bpsimple-# CAST(sell_price AS FLOAT) AS price FROM item;
CREATE

bpsimple=#
```

Если теперь выбирать данные из представления при помощи оператора SELECT, оно будет вести себя как подмножество столбцов исходной таблицы:

```
bpsimple=# SELECT * FROM item_price;
 item_id |  description  | price
---------+---------------+-------
       1 | Wood Puzzle   | 21.95
       2 | Rubik Cube    | 11.49
       3 | Linux CD      |  2.49
       4 | Tissues       |  3.99
       5 | Picture Frame |  9.95
       6 | Fan Small     | 15.75
       7 | Fan Large     | 19.95
       8 | Toothbrush    |  1.45
       9 | Roman Coin    |  2.45
      10 | Carrier Bag   |     0
      11 | Speakers      | 25.32
(11 rows)

bpsimple=#
```

Как это работает

Создано представление, содержащее только те столбцы таблицы, которые мы хотим сделать доступными пользователям. Приведем столбец sell_price к типу с плавающей точкой, для чего прибегнем к оператору CAST. Обратите внимание, что столбцу присвоено имя (посредством AS price) и пользователю виден именованный столбец.

Не существует ограничения, запрещающего использование в представлении нескольких таблиц. Можно записать сколь угодно сложное выражение на SQL и получить доступ к любому количеству таблиц.

Попробуйте сами. Создание представления из нескольких таблиц

Давайте создадим представление, которое решило бы проблему предъявления таблиц item и barcode в упрощенном виде, без информации о цене и без разделения данных на две таблицы. Вызываем команду CREATE VIEW all_items:

```
bpsimple=# CREATE VIEW all_items AS SELECT i.item_id, i.description,
bpsimple-# b.barcode_ean FROM item i, barcode b WHERE i.item_id =
b.item_id;
CREATE

bpsimple=#
```

Создается новое представление, с которым впоследствии можно будет работать как с таблицей:

```
bpsimple=# SELECT * FROM all_items;

 item_id |  description  |  barcode_ean
---------+---------------+---------------
       1 | Wood Puzzle   | 6241527836173
       2 | Rubik Cube    | 6241574635234
       3 | Linux CD      | 6264537836173
       3 | Linux CD      | 6241527746363
       4 | Tissues       | 7465743843764
       5 | Picture Frame | 3453458677628
       6 | Fan Small     | 6434564564544
       7 | Fan Large     | 8476736836876
       8 | Toothbrush    | 6241234586487
       8 | Toothbrush    | 9473625532534
       8 | Toothbrush    | 9473627464543
       9 | Roman Coin    | 4587263646878
      11 | Speakers      | 9879879837489
      11 | Speakers      | 2239872376872
(14 rows)

bpsimple=#
```

Заметьте, что абсолютно то же самое получится, если выполнить:

```
SELECT i.item_id, i.description, b.barcode_ean FROM item i, barcode b WHERE
i.item_id = b.item_id;
```

Наличие представления избавляет конечных пользователей от излишних сложностей.

Для вывода перечня представлений, существующих в базе данных, выполним команду `psql \dv`. А команда `\d` *название-представления* позволяет вывести описание представления и увидеть примененное для его создания выражение:

```
bpsimple=# \dv
    List of relations
   Name    | Type | Owner
-----------+------+-------
 all_items | view | rick
(1 row)

bpsimple=# \d all_items
               View "all_items"
  Attribute  |         Type          | Modifier
-------------+-----------------------+----------
 item_id     | integer               |
 description | character varying(64) |
 barcode_ean | character(13)         |
```

```
View definition: SELECT i.item_id, i.description, b.barcode_ean FROM item i,
barcode b WHERE (i.item_id = b.item_id);

bpsimple=#
```

Как это работает

Создано представление (оно получило название `all_items`), которое ведет себя как таблица во всем, кроме того, что данные для него берутся из некоего скрытого от пользователя SQL-выражения.

Некоторые программисты склонны считать, что представления настолько хороши, что надо скрыть под ними все таблицы. Но хотя в некоторых случаях представление действительно может быть удачным выходом из положения, в целом оно не настолько эффективно, насколько реальная таблица, в особенности если представление определяется сложным выражением SQL, а выборка данных осуществляется из нескольких таблиц. Если все данные «спрятаны» под представлениями, то производительность может снизиться, а пользователи не смогут оптимизировать выполнение своих SQL-выражений (например, если нужный им столбец находится в представлении, реализующем объединение большого количества таблиц). Даже если пользователю нужен всего один столбец, но он вынужден работать с представлением, то выполнено будет все сложное SQL-выражение (создающее представление), а это снизит производительность. Представления могут быть очень полезны, но сколько полезных вещей приносят еще и вред!

Представления уничтожаются аналогично таблицам:

```
DROP VIEW view-name
```

Однако, в отличие от случая с таблицами, уничтожение представления не влияет на данные, содержавшиеся в нем.

Внешние ключи

Пришло время изучить одну из самых важных разновидностей ограничений – **внешние ключи**.

При построении схемы учебной базы данных `bpsimple` мы видели, что таблицы с данными объединены или связаны с другими таблицами. Приведем в качестве напоминания схему отношений из главы 2 (рис. 8.1).

Мы уже знаем, как столбцы одной таблицы могут быть связаны со столбцами другой. Например, существует отношение между столбцом `customer_id` таблицы `orderinfo` и столбцом `customer_id` таблицы `customer`. Так, зная `orderinfo_id`, можно использовать `customer_id` из той же строки, чтобы узнать фамилию и адрес клиента, с которым связан данный заказ. Было рассказано о том, что `customer_id` является первичным ключом таблицы `customer`, то есть однозначно идентифицирует каждую

отдельную строку таблицы customer. Введем теперь еще один важнейший термин. Для таблицы orderinfo столбец customer_id является **внешним ключом**. Что означает такое понятие? Хотя customer_id и не является первичным ключом таблицы orderinfo, но зато столбец в таблице customer, с которым он соединен, уникален. Обратите внимание, что обратного отношения не существует: никакой из столбцов таблицы customer не является уникальным ключом какой-то другой таблицы. Следовательно, мы говорим, что таблица customer не имеет **внешних ключей**. При попытке создать ограничение внешний ключ PostgreSQL проверяет, объявлен ли столбец в конкретной таблице так, что он должен быть уникальным. Общераспространенной является ситуация, когда столбец, на который ссылается внешний ключ, представляет собою первичный ключ другой таблицы.

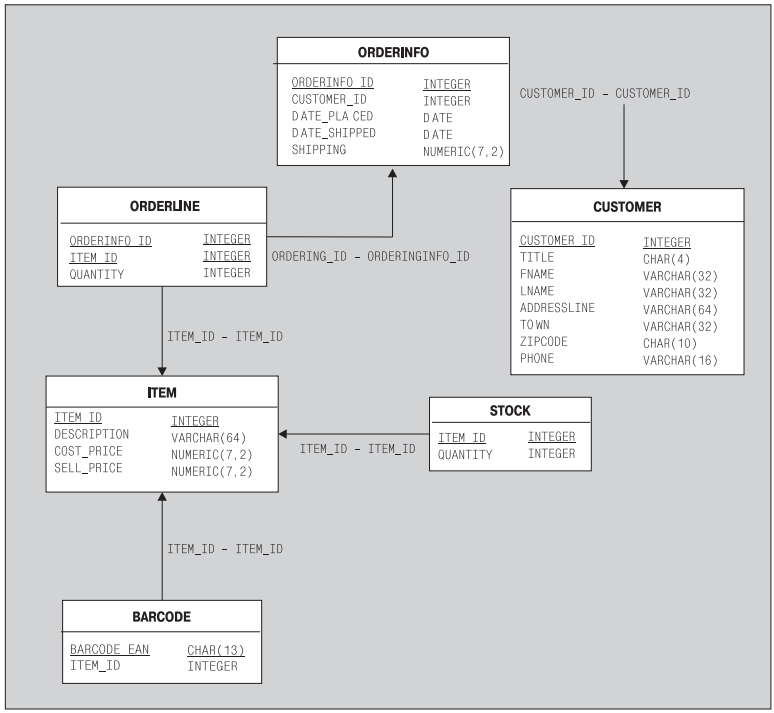


Рис. 8.1. Схема отношений базы данных bpsimple

Таблица может не иметь внешних ключей (как таблица customer), а может иметь и несколько. Так, в таблице orderline внешним ключом является orderinfo_id, поскольку он соединен с orderinfo_id – первичным ключом таблицы orderinfo. Но также внешним ключом является и item_id, который соединен с item_id таблицы item – первичным ключом этой таблицы.

В таблице item столбец item_id является ее первичным ключом, поскольку он однозначно идентифицирует строки, кроме того, он явля-

ется внешним ключом таблицы stock. Ситуация, когда один столбец представляет собою одновременно и первичный и внешний ключ, совершенно нормальна, она подразумевает (обычно необязательное) отношение «один-к-одному» между строками двух таблиц.

В нашей базе данных нет такого примера, но может быть и так, что пара столбцов, комбинация которых составляет внешний ключ (как orderinfo_id и item_id), является первичным ключом таблицы (orderline).

Такие отношения жизненно важны для нашей базы данных. Если в таблице orderinfo есть строка, в которой customer_id не соответствует customer_id таблицы customer, то это серьезная проблема. Есть заказ, и никакой информации о том, что за клиент его разместил. Конечно, можно использовать логику приложения для усиления правил, налагаемых на отношения (с помощью ограничений для столбцов и таблиц, которые упоминались ранее), но гораздо надежнее и часто проще объявить их как правила для базы данных.

Вряд ли вы удивитесь, узнав, что можно объявлять такие отношения между столбцами (внешние ключи), как ограничения для столбцов и таблиц, во многом подобно тем ограничениям, которые рассматривались ранее. Осуществляется такая операция при создании таблиц, для этого в команде CREATE TABLE указывается ограничение REFERENCES, обсуждение которого было отложено.

Давайте теперь оставим базу данных bpsimple и создадим базу bpfinal с использованием внешних ключей для обеспечения целостности данных.

Внешний ключ как ограничение для столбца

Приведем базовый синтаксис объявления столбца внешним ключом другой таблицы:

```
[CONSTRAINT arbitrary-name] existing-column-name type REFERENCES foreign-
table-name(column-in-foreign-table)
```

Присвоение ограничению имени не обязательно, но, как мы увидим позже, может облегчить процесс понимания сообщений об ошибках. Описание всех возможностей, всей мощи внешних ключей выходит за рамки этой книги. Для разрешения насущных проблем (пока вы не станете очень квалифицированным специалистом) должно хватить представленного стандартного синтаксиса. Более подробную информацию можно найти в оперативной справке.

Чтобы определить ограничение – внешний ключ для столбца customer_id таблицы orderinfo, связав его с таблицей customer, используйте ключевое слово REFERENCES вместе с названием внешней таблицы и столбца, а именно:

```
create table orderinfo
(
```

```
    orderinfo_id        serial,
    customer_id         integer not null REFERENCES customer(customer_id),
    date_placed         date not null,
    date_shipped        date,
    shipping            numeric(7,2) ,
    CONSTRAINT          orderinfo_pk PRIMARY KEY(orderinfo_id)
);
```

Заметьте, что ограничение получило название `orderinfo_pk`.

Вскоре вы увидите ограничение `REFERENCES` в действии.

Внешний ключ как ограничение для таблицы

Можно объявлять внешние ключи на уровне столбцов, но авторы предпочитают делать это на уровне таблиц, вместе с такими ограничениями, как `PRIMARY KEY`. Ограничение для столбца нельзя использовать, если в отношение вовлечено несколько столбцов текущей таблицы, в таких случаях приходится задавать ограничение для таблицы.

Вместо того чтобы смешивать внешние ключи, определенные как ограничения для столбцов и для таблиц, рекомендуется всегда использовать табличную форму.

Запись внешнего ключа для таблицы очень похожа на запись для столбца с той разницей, что указывается перечень всех столбцов:

```
[CONSTRAINT arbitrary-name] FOREIGN KEY (column-list) REFERENCES foreign-
table-name(column-list-in-foreign-table)
```

Можно обновить описание таблицы `orderinfo`, чтобы объявить ограничение: столбец `customer_id` представляет собою внешний ключ, т. к. он связан со столбцом `customer_id`, являющимся первичным ключом таблицы `customer`.

```
create table orderinfo
(
    orderinfo_id                    serial ,
    customer_id                     integer not null,
    date_placed                     date not null,
    date_shipped                    date ,
    shipping                        numeric(7,2) ,
    CONSTRAINT                      orderinfo_pk PRIMARY KEY(orderinfo_id),
    CONSTRAINT orderinfo_customer_id_fk FOREIGN KEY(customer_id) REFERENCES
         customer(customer_id)
);
```

Если удалить и пересоздать таблицу `orderinfo` (не забывая об автоматически сгенерированной последовательности), а затем снова ввести в нее данные, можно увидеть результат действия нового ограничения:

```
bpfinal=# DROP SEQUENCE orderinfo_orderinfo_id_seq;
DROP

bpfinal=# DROP TABLE orderinfo;
DROP

bpfinal=# CREATE TABLE orderinfo
bpfinal-# (
bpfinal-#     orderinfo_id                    serial,
bpfinal-#     customer_id                     integer not null,
bpfinal-#     date_placed                     date not null,
bpfinal-#     date_shipped                    date,
bpfinal-#     shipping                        numeric(7,2),
bpfinal-#     CONSTRAINT                      orderinfo_pk PRIMARY
bpfinal-#     KEY(orderinfo_id),
bpfinal-#     CONSTRAINT orderinfo_customer_id_fk FOREIGN KEY(customer_id)
bpfinal-#     REFERENCES customer(customer_id)
bpfinal-# );
NOTICE:  CREATE TABLE will create implicit sequence
'orderinfo_orderinfo_id_seq' for SERIAL column 'orderinfo.orderinfo_id'
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'orderinfo_pk'
for
table 'orderinfo'
NOTICE:  CREATE TABLE will create implicit trigger(s) for FOREIGN KEY
check(s)
CREATE

bpfinal=#
```

Посмотрев внимательно, можно увидеть дополнительное примечание `NOTICE` для команды `CREATE`, использованной ранее в данной книге:

```
CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
```

Сообщается, что созданы средства базы данных (в данном случае триггеры) для дополнительных проверок.

Теперь можно заново заполнить таблицу `orderinfo` при помощи имеющегося сценария SQL:

```
bpfinal=# \i pop_orderinfo.sql
INSERT 20325 1
INSERT 20326 1
INSERT 20327 1
INSERT 20328 1
INSERT 20329 1

bpfinal=#
```

Вернулись практически туда же, откуда начали, с той лишь (немаловажной) разницей, что теперь на таблицу `orderinfo` наложено ограничение – внешний ключ, который свидетельствует о том, что в строках таблицы `orderinfo` столбец `customer_id` связан со столбцом `customer_id`

таблицы customer. Это означает, что нельзя удалять строки из таблицы customer, если на строку ссылается столбец таблицы orderinfo.

Попробуйте сами. Внешние ключи

Для начала проверим, какие номера customer_id присутствуют в таблице orderinfo:

```
bpsimple=# select orderinfo_id, customer_id from orderinfo;
 orderinfo_id | customer_id
--------------+-------------
            1 |           3
            2 |           8
            3 |          15
            4 |          13
            5 |           8
            6 |           8
(6 rows)

bpsimple=#
```

Теперь известно, что в шести строках orderinfo столбец customer_id ссылается на клиентов из таблицы customer и что клиенты, на которых ссылаются, имеют идентификационные номера 3, 8, 13 и 15. Клиентов только четыре, потому что три строки (с orderinfo_id, равным 2, 5 и 6) связаны с одним и тем же клиентом.

Попробуем удалить из таблицы customer строку с customer_id, равным 3:

```
bpfinal=# DELETE FROM customer WHERE customer_id = 3;
ERROR:  orderinfo_customer_id_fk referential integrity violation - key in
customer still referenced from orderinfo

bpfinal=#
```

PostgreSQL не допускает удаления строки. Обратите внимание, благодаря тому что ограничение имеет название (orderinfo_customer_id_fk), проще идентифицировать источник проблемы. Если бы ограничение было безымянным, то PostgreSQL вывела бы просто <unnamed>. PostgreSQL позволит удалить строки из таблицы customer, только если не существует связанных с ними записей в orderinfo:

```
bpfinal=# DELETE FROM customer WHERE customer_id = 4;
DELETE 1

bpfinal=#
```

Как это работает

PostgreSQL работает «за кулисами», вводя некоторые дополнительные проверки, гарантируя, что для каждой строки таблицы customer,

которую попробуют удалить, будет проверено, не ссылается ли на нее строка другой таблицы (в данном случае – таблицы orderinfo).

Любая попытка нарушить это правило приведет к отклонению запроса на исполнение команды, и данные останутся неизмененными. Конечно же, удалить клиента можно, следует лишь предварительно убедиться, что у него нет заказов.

PostgreSQL контролирует и попытки вставить (посредством INSERT) в таблицу orderinfo строки, которые относятся к несуществующим клиентам:

```
bpfinal=# INSERT INTO orderinfo(customer_id, date_placed, shipping)
VALUEFS(250, '07-25-2000', 0.00);
ERROR:  orderinfo_customer_id_fk referential integrity violation - key
referenced from orderinfo not found in customer

bpfinal=#
```

Важно осознать, что таким образом был сделан значительный шаг вперед. Предприняты весьма эффективные меры, обеспечивающие установление отношений между таблицами на уровне базы данных. В таблице orderinfo больше не может быть строк, ссылающихся на несуществующих клиентов.

Теперь можно обновить сценарий create_tables.sql, добавив внешние ключи всем таблицам, которые ссылаются на другие таблицы, то есть orderinfo, orderline, stock и barcode.

Чуть сложнее ситуация с таблицей orderline, поскольку в ней столбец orderinfo_id ссылается на таблицу orderinfo, а столбец item_id – на таблицу item. Однако и здесь не возникает проблем, просто зададим два ограничения, по одному для каждого столбца, следующим образом:

```
create table orderline
(
    orderinfo_id                integer             not null,
    item_id                     integer             not null,
    quantity                    integer             not null,
    CONSTRAINT                  orderline_pk PRIMARY KEY(orderinfo_id,
item_id),
   CONSTRAINT orderline_orderinfo_id_fk FOREIGN KEY(orderinfo_id) REFERENCES
orderinfo(orderinfo_id),
    CONSTRAINT orderline_item_id_fk FOREIGN KEY(item_id) REFERENCES
item(item_id)
);
```

Полный набор ограничений для всех таблиц можно найти в наборе сценариев, который можно загрузить с веб-сайта издательства.

Работая с рассматриваемой базой данных, вы обнаружите, что данные в таблицы должны вводиться в порядке, обеспечивающем возмож-

ность соблюдения ограничений – внешних ключей, то есть теперь уже нельзя заполнить таблицу orderinfo до того, как будет заполнена customer, поскольку заказы, содержащиеся в первой таблице, ссылаются на вторую.

Предлагаем такой порядок ввода данных в таблицы:

- customer
- item
- orderinfo
- orderline
- stock
- barcode

Параметры внешних ключей

Сделаем еще один шаг вперед в проверках ссылочной целостности с помощью внешних ключей (это вопрос для более серьезного обсуждения, поэтому коснемся его лишь вкратце). Более полная информация представлена в документации из оперативной справки и книгах по SQL для опытных пользователей.

Может случиться так, что в таблице orderinfo есть записи, ссылающиеся на таблицу customer, при этом возникает необходимость обновить customer_id. Сделать это непросто, т. к. при попытке изменить customer_id (на самом деле это очень плохая мысль, ведь столбец имеет тип SERIAL!) внешний ключ таблицы orderinfo не допустит изменения, поскольку правила указывают, что customer_id, хранимый в каждой строке orderinfo, всегда должен ссылаться на запись customer_id таблицы customer.

Нельзя изменить customer_id в таблице orderinfo, потому что запись в таблице customer еще не существует, а запись в таблице customer невозможно изменить, потому что на нее ссылается таблица orderinfo.

Отложенные операции

Стандартный SQL предоставляет два пути выхода из подобной ситуации. Первый заключается в добавлении ключевого слова DEFERRABLE в конец внешнего ключа, а именно:

```
create table orderinfo
(
    orderinfo_id                serial ,
    customer_id                 integer not null,
    date_placed                 date not null,
    date_shipped                date ,
    shipping                    numeric(7,2) ,
```

```
    CONSTRAINT                    orderinfo_pk PRIMARY KEY(orderinfo_id),
    CONSTRAINT orderinfo_customer_id_fk FOREIGN KEY(customer_id) REFERENCES
customer(customer_id)DEFERRABLE
);
```

Изменяется способ введения в действие внешнего ключа. Обычно PostgreSQL проверяет соответствие внешним ключам перед тем, как разрешить какое бы то ни было изменение в базе данных. Если же применяются транзакции (поговорим о них в главе 9) и ключевое слово DEFERRABLE, то PostgreSQL позволит нарушить внешние ключи, но только внутри транзакции. Как мы узнаем позже, транзакция – это группа команд SQL, которая должна или выполниться полностью, или же все должно остаться так, как будто ни одна из команд не была выполнена. Следовательно, можно начать транзакцию, обновить customer_id в таблице customer, обновить соответствующие номера customer_id в таблице orderinfo, фиксировать транзакцию, и тогда PostgreSQL разрешит совершить такие действия. Внешние ключи будут проверены после окончания транзакции.

ON UPDATE и ON DELETE

Альтернативным решением является указание во внешнем ключе правил, регулирующих обработку нарушений в двух случаях – при обновлении (UPDATE) и при удалении (DELETE). Возможны два действия. Во-первых, можно каскадировать (CASCADE) изменение из таблицы с первичным ключом, во-вторых, можно установить столбец в NULL (SET NULL), т. к. он больше не ссылается на исходную таблицу.

Например:

```
create table orderinfo
(
    orderinfo_id                  serial ,
    customer_id                   integer not null,
    date_placed                   date not null,
    date_shipped                  date ,
    shipping                      numeric(7,2) ,
    CONSTRAINT                    orderinfo_pk PRIMARY KEY(orderinfo_id),
    CONSTRAINT orderinfo_customer_id_fk FOREIGN KEY(customer_id) REFERENCES
customer(customer_id) ON DELETE CASCADE
);
```

Здесь вы сообщаете PostgreSQL, что если удаляется строка таблицы customer, имеющая customer_id, который используется в таблице orderinfo, то PostgreSQL должна автоматически удалить соответствующие строки из orderinfo. Может быть, это именно то, что нужно, но вообще-то такой выбор опасен. В большинстве случаев лучше обеспечить удаление приложениями строк в правильном порядке, чтобы иметь возможность убедиться, что для клиента не существует заказов, прежде чем удалять клиентскую запись.

Параметр SET NULL обычно применяется для обновлений (UPDATE); выглядит это так:

```
create table orderinfo
(
    orderinfo_id                    serial ,
    customer_id                     integer not null,
    date_placed                     date not null,
    date_shipped                    date ,
    shipping                        numeric(7,2) ,
    CONSTRAINT                      orderinfo_pk PRIMARY KEY(orderinfo_id),
    CONSTRAINT orderinfo_customer_id_fk FOREIGN KEY(customer_id) REFERENCES
customer(customer_id) ON UPDATE SET NULL
);
```

Сообщается, что если следует удалить из таблицы customer строку, на которую ссылается customer_id, то столбец таблицы orderinfo устанавливается в NULL. Внимательные читатели должны были заметить, что в нашей таблице этот способ не работает. Столбец customer_id объявлен как NOT NULL, поэтому нельзя изменить его значение на NULL. Дело в том, что нельзя допустить, чтобы строки таблицы orderinfo имели в столбце неизвестный customer_id. Ведь как можно трактовать заказ от неизвестного клиента? Вероятнее всего, как ошибку.

Параметры можно комбинировать, например:

```
ON UPDATE SET NULL ON DELETE CASCADE
```

Данный раздел (ON UPDATE и ON DELETE) включен в описание для полноты картины. Однако советуем использовать параметры с осторожностью. Гораздо надежнее заставить разработчиков приложений запрограммировать операторы UPDATE и DELETE в правильном порядке и использовать транзакции, чем применять к строкам CASCADE DELETE и вдруг ни с того ни с сего сохранять в столбцах NULL значения из-за того, что была изменена другая таблица.

В главе 10 «Хранимые процедуры и триггеры» будет рассказано о том, что с помощью триггеров и хранимых процедур можно добиться практически такого же результата, при этом получая больше возможностей для контроля над изменениями других таблиц.

Резюме

В данной главе более формально были описаны типы данных, поддерживаемые PostgreSQL, главным образом это были стандартные типы SQL, но упоминались и некоторые специальные типы PostgreSQL, такие как массивы.

Затем рассматривалась обработка данных в столбцах, приведение типов, выделение частей строк, а также несколько «магических» переменных, разрешающих доступ к информации, например CURRENT_USER.

Потом была представлена такая важная тема, как ограничения. Рассказано о двух способах определения ограничений: для отдельного столбца и для всей таблицы в целом. Было показано, как даже с помощью простых ограничений можно ввести в действие правила целостности на уровне базы данных.

Последними обсуждались внешние ключи – один из самых важных видов ограничений, позволяющий формально определить отношения между таблицами в базе данных. Самое ценное заключается в том, что внешние ключи позволяют «навязать» правила базе данных, гарантируя таким образом, например, что никогда не будет удален клиент, если в другой таблице есть информация о заказах, связанных с данным клиентом.

9

Транзакции и блокировки

До сих пор мы всячески избегали разговоров о многопользовательских аспектах работы PostgreSQL, довольствуясь идеализированным представлением, то есть полагая, что (как и любая другая хорошая реляционная база данных) PostgreSQL скрывает детали поддержки нескольких одновременно работающих пользователей. Она просто предоставляет сервер базы данных, который работает так, словно все одновременно работающие пользователи, не снижая эффективности сервера, имеют монопольный доступ и каждый из них независим от остальных.

Во многих случаях, в частности для небольших и малозагруженных баз, это идеализированное представление почти достижимо на практике. Реальность заключается в том, что PostgreSQL, несмотря на все ее возможности, все-таки не волшебница. Изоляция каждого отдельного пользователя от всех остальных требует большой закулисной работы. Порой реальная жизнь грубо меняет идеализированные представления пользователей, считающих, что им предоставлен исключительный доступ к серверу базы данных.

В этой главе основное внимание будет уделено не способам, которыми PostgreSQL достигает разделения пользователей, а практическим сторонам этой ситуации для пользователя базы данных. Поговорим о том, какого поведения могут ожидать от базы данных клиентские приложения и как они должны работать с сервером, чтобы максимально увеличить свою производительность. Будет показано, как PostgreSQL предоставляет возможность собрать несколько дискретных изменений базы данных в единый рабочий период – транзакцию. Такая возможность чрезвычайно важна, когда необходимо выполнить набор изменений как единое целое.

Будут рассматриваться:

- Основные сведения о транзакциях
- Свойства ACID
- Транзакции для одного пользователя
- Ограничения на применение транзакций
- Транзакции для нескольких пользователей
- Уровни изоляции ANSI
- Режимы явных и неявных транзакций
- Блокировки
- Взаимные и явные блокировки

Что такое транзакция?

Первое, что следует обсудить – каким образом в базу данных вносятся изменения. Уже говорилось, что (всегда, когда возможно) следует выполнять изменения в базе данных одним декларативным оператором, но при работе с реальными приложениями случается так, что требуется произвести несколько изменений, которые не могут быть записаны в одном операторе SQL. Но по-прежнему будет необходимо, чтобы были сделаны либо все эти изменения, либо же ни одно из них (если с любым из них возникают проблемы).

Рассмотрим классический пример: перевод денег с одного банковского счета на другой, при этом счета могут быть представлены в разных таблицах базы данных, сумма должна быть внесена в дебет одного и в кредит другого счета. Если первый счет будет дебетован, а зачисление суммы в кредит второго счета не удастся, то следует вернуть деньги на первый счет или же сделать вид, что деньги никогда не снимались с него. Если банк «теряет» деньги при переводе со счета на счет, он должен прекратить свою деятельность!

В базах данных, построенных на ANSI SQL, в том числе PostgreSQL, для решения вышеописанной задачи применяются так называемые **транзакции**.

> ***Транзакция – это логическая единица работы, которая не должна разделяться на части.***

Что понимается под логической единицей работы? Это просто набор логических изменений базы данных, которые должны или произойти все вместе, или же ни одно из них. Все как в примере с переводом денег. В PostgreSQL такие изменения контролируются тремя ключевыми фразами:

- `BEGIN WORK`, начинающей транзакцию.

- COMMIT WORK, сообщающей, что все элементы транзакции завершены и теперь должны быть сделаны перманентными и доступными всем параллельным и последующим транзакциям.
- ROLLBACK WORK, означающей, что транзакция должна быть ликвидирована, а все изменения данных, произведенные операторами такой транзакции, отменяются. Для всех пользователей база данных должна выглядеть так, как будто после предыдущего BEGIN WORK никаких изменений сделано не было.

В стандартном ANSI/ISO SQL нет фразы BEGIN WORK, транзакция определена как начинающаяся автоматически (то есть эта фраза будет избыточной), но она общепринята и необходима во многих реляционных базах данных.

Как в** ROLLBACK WORK**, так и в** COMMIT WORK **часть** WORK **может быть опущена.

Вторым аспектом транзакции является то, что каждая транзакция в базе данных изолирована ото всех остальных транзакций, происходящих в базе данных в то же самое время. В идеале каждая транзакция должна была бы вести себя так, как будто ей предоставлен полный доступ к базе данных. К сожалению, как мы увидим позже в этой же главе, при рассмотрении уровней изоляции выполняемых операций окажется, что в реальной жизни для достижения высокой производительности часто приходится идти на компромисс. Рассмотрим еще один пример, требующий использования транзакции.

Предположим, что необходимо зарезервировать авиабилет через Интернет. Проверяем интересующий нас рейс и выясняем, что билет есть. Мы этого не знаем, но на самом деле это последний билет на данный рейс. Пока мы вводим информацию о кредитной карте, еще один клиент, имеющий счет в авиакомпании, интересуется тем же самым рейсом. Билет еще не приобретен нами, поэтому выводится свободное посадочное место, которое тот клиент и резервирует, пока мы продолжаем вводить свои данные. Затем мы передаем данные для покупки «нашего» билета, а поскольку система знает, что в начале транзакции в самолете было свободное место, то она ошибочно считает, что место все еще свободно и снимает деньги с кредитной карты.

Мы отсоединяемся от системы в полной уверенности, что место зарезервировано и, вероятно, проверяем, сняты ли деньги с кредитной карты. Правда же в том, что куплен несуществующий билет. В момент обработки нашей транзакции свободных мест уже не оставалось.

Порядок действий, выполняемых при продаже билетов, в данном случае выглядит примерно так:

```
Проверить наличие свободных мест.
Если место есть, предложить его клиенту.
Если клиент принимает предложение, то запросить номер его кредитной карты.
Провести авторизацию кредитной карты в банке.
```

```
Дебетовать нужную сумму.
Отвести клиенту место.
Уменьшить количество свободных мест на количество приобретенных мест.
```

Такая последовательность событий абсолютно обоснованна в случае, если в каждый момент времени только один клиент работает с системой. Неприятности объясняются наличием двух клиентов. Вот что произошло в действительности (табл. 9.1):

Таблица 9.1. Что такое транзакция

Клиент 1	Клиент 2	Свободные места в самолете
Есть ли свободные места?		1
	Есть ли свободные места?	1
Если да, предложить место клиенту		1
	Если да, предложить место клиенту	1
Если клиент согласен, запросить номер кредитной карты или расчетного счета		1
	Если клиент согласен, запросить номер кредитной карты или расчетного счета	1
Вводит номер карты	Вводит номер счета	1
Авторизация кредитной карты в банке		1
	Проверка действительности счета	1
	Обновить сумму на счете (новая транзакция)	1
Дебетование карты	Выделение места	1
Выделение места	Уменьшить количество свободных мест на число приобретенных	0
Уменьшить количество свободных мест на число приобретенных		-1

Положение дел существенно поправилось бы, если бы еще одна проверка доступности билетов осуществлялась ближе к моменту снятия денег. Но как бы близко не удалось подойти, шаг «проверка доступности мест» неизбежно отделен от шага «снятие денег», даже если их и разделяет самый маленький временной промежуток.

Для решения проблемы можно броситься в другую крайность, разрешив в каждый момент времени доступ только одного пользователя в систему резервирования, но производительность бы катастрофически упала и клиенты ушли бы в другое место. Есть еще альтернатива. Можно написать приложение с применением семафора или другой подобной технологии для управления доступом к критическим точкам программы. Тогда каждое приложение, обращающееся к базе данных, должно было бы использовать семафор, такой способ намного эффективнее и лучше подходит для решения проблемы. Весьма маловероятно, что в какой-то авиакомпании применяется подобная упрощенная система, в которой происходят ошибки при бронировании билетов, но такой пример удобен для иллюстрации принципа.

В терминах приложения можно сказать, что существует критическая часть программы, маленький фрагмент кода, которому необходим эксклюзивный доступ к некоторым данным. Как мы уже видели, часто проще разрешать проблемные ситуации при помощи базы данных, а не заниматься написанием логики для приложения. Теперь опишем обстановку в терминах базы данных: присутствует транзакция, набор операций с данными, начинающийся проверкой свободных мест и заканчивающийся дебетованием кредитной карты или счета и выделением места; при этом все операции должны рассматриваться как одна единица работы.

Свойства АСИД

АСИД (ACID) – это часто используемое обозначение, описывающее свойства, которыми должна обладать транзакция:

- A – **Атомарность** (Atomicity)

  Несмотря на то что транзакция является набором операций, она должна происходить как единое целое. Транзакция должна осуществляться ровно один раз, в ней нет никаких подмножеств. В рассмотренном банковском примере перемещение должно быть элементарным. Должно быть выполнено как дебетование одного счета, так и кредитование другого, как если бы это была единая операция, даже если при этом необходимо выполнение нескольких последовательных операторов SQL.

- C – **Согласованность** (Consistency)

  По окончании транзакции система должна находиться в непротиворечивом состоянии. Эта тема затрагивалась в главе 8, там было показано, что ограничение можно объявить как допускающее задержку (deferrable). Другими словами, ограничение должно проверяться лишь после завершения транзакции. В примере с банком в конце транзакции на всех счетах должны быть правильные суммы.

- И – **Изоляция** (Isolation)

  Каждая транзакция вне зависимости от того, сколько транзакций в данный момент выполняются в базе данных, должна быть независимой ото всех остальных транзакций. В примере с авиакомпанией транзакции, обслуживающие двух одновременно работающих клиентов, должны вести себя так, как если бы каждая имела исключительный доступ к базе данных. На практике, как мы знаем, это невозможно (если стремиться к разумной производительности для многопользовательских баз данных). Это как раз одно из тех мест, где суровая реальность наиболее сильно влияет на идеализированное поведение базы данных. Вернемся к теме изолированности чуть позже в данной главе.

- Д – **Долговечность** (Durability)

  После того как транзакция завершена, она должна оставаться в этом состоянии. После успешного перевода со счета на счет деньги должны оставаться переведенными даже в случае внезапного выключения питания на компьютере, на котором работает СУБД. В PostgreSQL, как и в большинстве реляционных баз данных, это достигается за счет ведения журнала транзакций. Этот процесс чрезвычайно прост. При выполнении транзакции изменения записываются не только в базу данных, но и в журнальный файл. По завершении транзакции записывается маркер, указывающий на конец транзакции, и журнал принудительно сохраняется на постоянное запоминающее устройство, поэтому он не подвергается опасности даже в случае аварийной ситуации на сервере базы данных. Если по какой-то причине сервер базы данных отключается посреди транзакции, то после перезапуска он может автоматически обеспечить корректное завершение транзакции в базе данных (завершив операции, записанные в журнале, но не выполненные в базе данных). Изменения, вносимые транзакциями, которые находились в процессе исполнения в момент отказа сервера, не будут отражены в базе данных. Это свойство транзакций не требует вмешательства пользователя, то есть не зависит от него, поэтому не будем больше заниматься его рассмотрением.

Транзакции при однопользовательском доступе

Прежде чем приступить к рассмотрению более сложных аспектов транзакции и их поведения в случае одновременного доступа нескольких пользователей к базе данных, стоит поговорить о том, как они ведут себя при работе с одним пользователем.

Даже при таком упрощенном способе работы применение транзакций имеет явные преимущества. Польза транзакций в том, что они позво-

ляют выполнить несколько операторов SQL, а затем, на более позднем этапе, при необходимости возвратиться к предыдущему состоянию, отменив проделанную работу. Если применяются транзакции, то приложение не должно беспокоиться о хранении информации об изменениях, вносимых в базу данных, и о том, как их отменить. Оно просто обращается к серверу базы данных, отменяющему всю группу изменений одним махом.

Последовательность операций представлена на рис. 9.1:

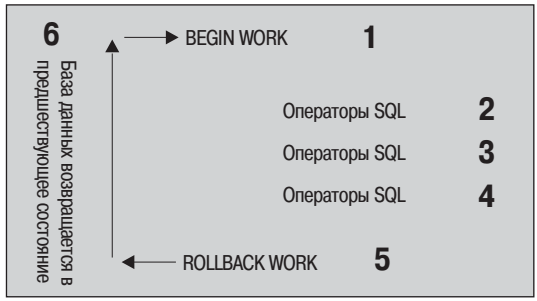


Рис. 9.1. Последовательность операций при использовании транзакции (отмена выполненных действий)

Если же на шаге 5 окажется, что все изменения базы данных верны и надо таки применить их, чтобы они стали постоянными, то достаточно заменить оператор `ROLLBACK WORK` на оператор `COMMIT WORK` (рис. 9.2):

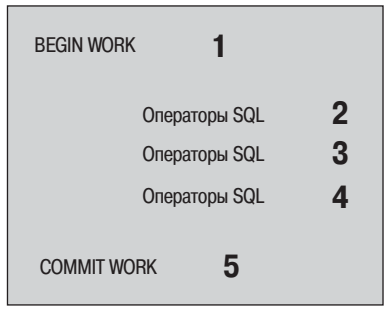


Рис. 9.2. Последовательность операций при использовании транзакции (фиксация выполненных действий)

После шага 5 изменения фиксируются в базе данных и могут считаться постоянными, поэтому они не будут утеряны при сбое питания, неисправности диска или в случае ошибки приложения.

Журнал транзакций, который ведет PostgreSQL, записывает не только все изменения, вносимые в базу данных, но также и способ их отмены. Очевидно, что такой файл может очень быстро вырасти до огромных размеров. Однако как только PostgreSQL встречает оператор `COMMIT`

WORK для транзакции, становится понятно, что хранить информацию об отмене операций больше не надо, т. к. теперь изменения базы данных стали необратимыми, по крайней мере, средствами СУБД. Конечно же, можно ввести в приложение дополнительный код для отмены сделанных изменений.

Попробуйте сами. Транзакции

Создадим самую простую транзакцию, в которой будет изменяться одна строка таблицы, имя David в ней будет заменено на Dave, а затем посредством оператора ROLLBACK отменим изменения:

```
bpfinal=# BEGIN WORK;
BEGIN
bpfinal=# SELECT fname FROM customer WHERE customer_id = 15;
fname
--------
David
(1 row)
bpfinal=# UPDATE customer SET fname = 'Dave' WHERE customer_id = 15;
UPDATE 1
bpfinal=# SELECT fname FROM customer WHERE customer_id = 15;
fname
--------
Dave
(1 row)
bpfinal=# ROLLBACK WORK;
ROLLBACK
bpfinal=# SELECT fname FROM customer WHERE customer_id = 15;
fname
--------
David
(1 row)
bpfinal=#
```

Как это работает

Транзакцию начнем командой BEGIN WORK. Затем вносим изменение в базу данных, обновив столбец fname в строке со значением customer_id, равным 15. Применим к данной строке SELECT и увидим, что данные изменены. После этого выполним ROLLBACK WORK. PostgreSQL использует свой внутренний журнал транзакций для отмены изменений, проделанных после того, как был выполнен оператор BEGIN WORK, поэтому когда данные выбираются (посредством SELECT) из строки с customer_id = 15 во второй раз, внесенное изменение уже отменено.

В PostgreSQL часть WORK операторов BEGIN WORK, COMMIT WORK и ROLLBACK WORK лишь создает неудобства, для PostgreSQL достаточно употребления ключевых слов BEGIN, COMMIT и ROLLBACK. В книге полная форма приводится для большей понятности.

Действие транзакций не ограничивается одной таблицей или простыми обновлениями данных. Приведем более сложный пример, в котором участвуют два оператора – UPDATE и INSERT, применяемые к разным таблицам:

```
bpfinal=# BEGIN WORK;
BEGIN
bpfinal=# INSERT INTO customer(title, fname, lname, addressline, town,
bpfinal-# zipcode, phone)  VALUES('Mr', 'Steven', 'Harvey', '23 Millbank
bpfinal-# Road', 'Nicetown', 'NT1 1EE', '267 4323');
INSERT 21099 1
bpfinal=# UPDATE item SET sell_price = 99.99 WHERE item_id = 2;
UPDATE 1
bpfinal=# ROLLBACK WORK;
ROLLBACK
bpfinal=# SELECT fname, lname FROM customer WHERE lname = 'Harvey';
 fname | lname
-------+-------
(0 rows)

bpfinal=# SELECT * FROM item WHERE item_id = 2;
 item_id | description | cost_price | sell_price
---------+-------------+------------+------------
       2 | Rubik Cube  |       7.45 |      11.49
(1 row)

bpfinal=#
```

Данные, добавленные в результате выполнения оператора INSERT, были удалены, а проведенное обновление (посредством UPDATE) таблицы item – отменено.

Ограничения использования транзакций

Есть несколько моментов, на которые стоит обратить внимание при использовании транзакций.

Во-первых, в PostgreSQL и многих других реляционных СУБД не допускается применение вложенных транзакций. В PostgreSQL при попытке выполнить оператор BEGIN WORK во время транзакции будет выведено уведомление о том, что транзакция уже находится в процессе исполнения. В некоторых СУБД существует понятие «точек сохранения», присутствующее в стандарте SQL. Это точки, в которых можно поставить метку внутри транзакции и осуществлять откат операций, произведенных до точки сохранения, а не всех операций транзакции.

Некоторые базы данных допускают наличие нескольких операторов BEGIN WORK, не выдавая никаких сообщений. Команда COMMIT WORK или ROLLBACK WORK всегда рассматривается по отношению к первому оператору BEGIN WORK, поэтому несмотря на то, что существует видимость вложенных транзакций, на самом деле последующие операторы BEGIN WORK игнорируются.

В настоящее время PostgreSQL не поддерживает точки сохранения, зато при попытке выполнить вложенную транзакцию при помощи `BEGIN WORK` в процессе исполнения транзакции выдает предупреждение. Поскольку на настоящий момент PostgreSQL не поддерживает точки сохранения, больше не будем рассматривать их в данной книге.

Во-вторых, не рекомендуется делать транзакции большими. Как будет показано далее в этой главе, PostgreSQL (и другие реляционные СУБД) должны проделывать массу работы для того, чтобы обеспечить разделение транзакций разных пользователей. В результате участки базы данных, вовлеченные в транзакции, должны часто блокироваться (что гарантирует разделение транзакций).

PostgreSQL блокирует базу данных автоматически, но длительная транзакция обычно не позволяет другим пользователям обратиться к данным, участвующим в транзакции, до тех пор пока она не будет закончена или отменена. Рассмотрим приложение, запускающее транзакцию, когда пользователь утром усаживается за свой терминал. Эта транзакция выполняется в течение всего дня, пока пользователь производит различные изменения в базе. Предположим, что изменения фиксируются, только когда пользователь покидает свое рабочее место в конце дня. В рассмотренном случае поведение пользователя оказывает очень сильное отрицательное влияние на производительность базы данных и возможность других пользователей получить доступ к данным.

Не следует также затевать диалог с пользователем в процессе выполнения транзакции. Рекомендуется сначала собрать все данные, а затем уже обрабатывать их в транзакции, освобожденной от непредсказуемых ответов пользователя.

Оператор `COMMIT WORK`, как правило, выполняется достаточно быстро, поскольку на самом деле ему не так уж много надо сделать. А вот откат транзакций обычно приводит к выполнению по меньшей мере такой же работы, как при их выполнении, а часто и гораздо большей. Следовательно, если после запуска транзакции, которая (а точнее, все ее SQL-операторы) выполнялась 5 минут, вы решили отменить все изменения с помощью `ROLLBACK WORK`, не ждите, что откат будет мгновенным. Отмена всех изменений вполне может занять больше 5 минут.

Транзакции при многопользовательском доступе

Прежде чем перейти к разговору об особенностях транзакций, обеспечивающих работу нескольких пользователей, и о том, как транзакции изолируются друг от друга, вернемся к правилам АСИД и рассмотрим более подробно свойство изоляции.

Уровни изоляции ANSI

Как уже говорилось, одним из наиболее сложных аспектов реляционных СУБД является изолированность различных пользователей, вносящих изменения в базу данных. Конечно, существует простой способ обеспечения изоляции. Разрешение единственного соединения с базой, выполнение только одной транзакции в каждый момент времени гарантирует полное разделение различных транзакций. Сложность заключается в поиске более практичного решения, не приводящего к значительному снижению производительности, препятствующему доступу к базе нескольких пользователей.

Добиться настоящей, полной изоляции без серьезного снижения производительности практически невозможно, поэтому стандарт ANSI/ISO SQL определяет несколько уровней изоляции выполняемых операций, которые могут быть реализованы в СУБД. Обычно в реляционных базах данных по умолчанию обеспечивается по меньшей мере один из этих уровней, и пользователю предоставляется возможность определить хотя бы еще один уровень изоляции по своему выбору.

Чтобы разобраться со стандартными уровнями изоляции, освоим сначала еще несколько понятий. Несмотря на то что в большинстве случаев можно довольствоваться поведением PostgreSQL по умолчанию, бывают ситуации, когда полезно понимать все детали происходящего.

Нежелательные явления

Стандарт ANSI/ISO SQL определяет уровни изоляции в терминах нежелательных явлений, возможных при взаимодействии транзакций в многопользовательских базах данных.

Неаккуратное считывание

Считывание данных называется неаккуратным (dirty reads), если какой-то оператор SQL, входящий в транзакцию, считывает данные, которые были изменены другой транзакцией, но при этом изменения, сделанные второй транзакцией, еще не зафиксированы.

Мы уже знаем, что транзакция – это логическая единица (или блок) работы, которая должна быть атомарной. Должны совершиться или все элементы, входящие в нее, или ни один из них. До тех пор пока транзакция не зафиксирована, всегда существует вероятность, что она не выполнится или же будет прекращена командой `ROLLBACK WORK`. Поэтому другие пользователи базы данных не должны видеть измененные данные до выполнения оператора `COMMIT`.

Проиллюстрируем вышесказанное на примере (табл. 9.2) и посмотрим, что разные транзакции могут увидеть в столбце `fname` для клиента со значением `customer_id`, равным `15`. Предполагаем при этом, что подобные ситуации (неаккуратное считывание) возможны, на самом же деле PostgreSQL не допускает этого, так что это лишь теоретический пример.

Таблица 9.2. Неаккуратное считывание

Транзакция 1	Данные, которые видит 1	То, что увидят другие транзакции при неаккуратном считывании	То, что увидят другие транзакции, если неаккуратного считывания не происходит
BEGIN WORK			
	David	David	David
UPDATE customer SET fname='Dave'			
WHERE customer_id = 15;			
	Dave	Dave	David
COMMIT WORK			
	Dave	Dave	Dave
BEGIN WORK			
UPDATE customer SET fname = 'David' WHERE customer_id = 15;			Dave
	David	David	Dave
ROLLBACK WORK			
	Dave	Dave	Dave

Как видите, в результате неаккуратного считывания другие транзакции увидели данные, которые еще не зафиксированы в базе данных. Другими словами, они могут видеть изменения, которые затем будут отменены командой ROLLBACK WORK.

PostgreSQL ни при каких обстоятельствах не допускает неаккуратного считывания.

Неповторяемое считывание

Данный процесс похож на неаккуратное считывание, но имеет более ограничительный смысл. Неповторяемое считывание (unrepeatable reads) происходит, если транзакция читает набор данных, потом заново прочитывает данные и обнаруживает, что они изменились. Это не такая серьезная проблема, как при неаккуратном считывании, но все же ситуация далека от идеала. На что это похоже, можно себе представить, просмотрев табл. 9.3:

Таблица 9.3. Неповторяемое считывание

Транзакция 1	Данные, которые видит 1	То, что увидят другие транзакции в результате неповторяемого считывания	То, что увидят другие транзакции, если неповторяемого считывания не происходит
BEGIN WORK		BEGIN WORK	BEGIN WORK
	David	David	David

Транзакция 1	Данные, которые видит 1	То, что увидят другие транзакции в результате неповторяемого считывания	То, что увидят другие транзакции, если неповторяемого считывания не происходит
UPDATE customer SET fname = 'Dave' WHERE customer_id = 15;			
	Dave	David	David
COMMIT WORK			
	Dave	Dave	David
		COMMIT WORK	COMMIT WORK
		BEGIN WORK	BEGIN WORK
SELECT fname FROM customer WHERE customer_id = 15;		Dave	Dave

Обратите внимание, что при неповторяемом считывании транзакция может видеть изменения, зафиксированные другими транзакциями, даже если сама читающая транзакция еще не зафиксирована. Если же неповторяемые считывания запрещены, то другие транзакции не видят изменений, произведенных в базе данных, до тех пор пока они не зафиксируют свои изменения.

По умолчанию PostgreSQL разрешает неповторяемые считывания, позже будет рассказано о том, как изменить поведение по умолчанию на более удобное.

Фантомное чтение (phantom reads)

Проблема очень похожа на только что описанные неповторяемые считывания, только она возникает при добавлении в таблицу новой строки в то время, когда другая транзакция обновляет таблицу, при этом новая строка должна быть обновлена, но этого не происходит.

Предположим, что две транзакции вносят изменения в таблицу `item`. Первая увеличивает отпускную цену на 1 доллар, а вторая добавляет новое изделие (табл. 9.4):

Таблица 9.4. Фантомное чтение

Транзакция 1	Транзакция 2
BEGIN WORK	BEGIN WORK
UPDATE item SET sell_price = sell_price + 1;	
	INSERT INTO item(….) VALUES(…);
COMMIT WORK	
	COMMIT WORK

Каким должно быть значение отпускной цены (значение sell_price) изделия, добавленного транзакцией 2? Оператор INSERT был запущен до того, как UPDATE был зафиксирован, поэтому разумно ожидать, что она окажется на единицу больше введенной величины. Если же фантомное чтение возможно, то новая запись, появляющаяся после того, как транзакция 1 определила, к каким строкам должен быть применен UPDATE, не изменяется и цена нового изделия не увеличивается.

Фантомное чтение имеет место чрезвычайно редко и его практически невозможно продемонстрировать, поэтому в большинстве случаев беспокоиться об этом не стоит, хотя по умолчанию PostgreSQL разрешает подобные считывания.

Потеря результатов обновления

Потеря результатов обновления несколько отличается от предыдущих трех описанных случаев, будучи, как правило, проблемой на уровне приложения, не связанной с образом действий реляционной базы данных. С другой стороны, потерянные изменения встречаются тогда, когда в базу данных записываются два различных обновления, и в результате второго первое оказывается утраченным.

Предположим, что два пользователя работают с интерактивным приложением, которое изменяет таблицу item (табл. 9.5):

Таблица 9.5. Потеря результатов обновления: приложение изменяет таблицу item

Пользователь 1		**Пользователь 2**	
Его действия	**Данные, которые он видит**	**Его действия**	**Данные, которые он видит**
Попытка изменить отпускную цену с 21.95 на 22.55		Попытка изменить себестоимость с 15.23 на 16.00	
BEGIN WORK		BEGIN WORK	
SELECT cost_price, sell_price from item WHERE item_id = 1;	15.23, 21.95	SELECT cost_price, sell_price from item WHERE item_id = 1;	15.23, 21.95
UPDATE item SET cost_price = 15.23, sell_price = 22.55 WHERE item_id = 1;			
	15.23, 22.55		
COMMIT WORK			
			15.23, 22.55
		UPDATE item SET cost_price = 16.00, sell_price = 21.95 WHERE item_id = 1;	
	15.23, 22.55		16.00, 21.95

Пользователь 1		Пользователь 2	
Его действия	**Данные, которые он видит**	**Его действия**	**Данные, которые он видит**
	16.00, 21.95	COMMIT WORK	16.00, 21.95

Изменение sell_price, сделанное пользователем 1, было утрачено, но не из-за ошибки в базе данных, а из-за того, что пользователь 2 прочитал значение sell_price, «подержал» его некоторое время, а затем записал обратно в базу, уничтожив изменение, сделанное пользователем 1. База данных вполне корректно изолировала два набора изменений, но, тем не менее, приложение потеряло данные.

Есть несколько способов решить эту проблему, но выбор наиболее подходящего зависит от конкретного приложения. В качестве первого шага приложения должны принимать меры для того, чтобы сделать транзакции как можно короче, никогда не позволяя им выполняться дольше, чем необходимо. Кроме того, приложения должны сразу записывать данные, которые были изменены. Выполнение этих двух условий предотвратит множество утраченных изменений, в том числе и ошибку, рассмотренную выше.

Конечно же, возможна ситуация, когда оба пользователя пытаются изменить sell_price, тогда изменение все равно будет утеряно. Универсальный способ препятствования потере изменений заключается в указании значения, которое нужно изменить, в операторе UPDATE (табл. 9.6):

Таблица 9.6. Потеря результатов обновления: оператор UPDATE

Пользователь 1		Пользователь 2	
Его действия	**Данные, которые он видит**	**Его действия**	**Данные, которые он видит**
Попытка изменить отпускную цену с 21.95 на 22.55		Попытка изменить отпускную цену с 21.95 на 22.99	
BEGIN WORK		BEGIN WORK	
Read sell_price where item_id = 1	21.95	Read sell_price where item_id = 1	21.95
UPDATE item SET cost_price = 15.23, sell_price = 22.55 WHERE item_id = 1 and sell_price = 21.95;			
	22.55		21.95
COMMIT WORK			
			22.55

Таблица 9.6 (продолжение)

Пользователь 1		Пользователь 2	
Его действия	**Данные, которые он видит**	**Его действия**	**Данные, которые он видит**
		`UPDATE item SET cost_price = 16.00, sell_price = 21.95 WHERE item_id = 1 and sell_price = 21.95;` Обновление не выполняется, т. к. строка не найдена, потому что значение `sell_price` изменено	

Данный способ нельзя считать панацеей, т. к. он работает, только если первая транзакция фиксируется (выполняется `COMMIT`) до того, как запускается второй `UPDATE`, однако он значительно снижает риск потери сделанных изменений.

Уровни изоляции ANSI/ISO

Итак, новые термины изучены, и пришло время познакомиться с тем, как стандарт ANSI/ISO определил различные уровни изоляции, которые может использовать база данных. Каждый из уровней ANSI/ISO представляет собой комбинацию трех первых типов нежелательного поведения, описанных ранее (табл. 9.7):

Таблица 9.7. Уровни изоляции ANSI/ISO

Определение уровня изоляции ANSI/ISO	Неаккуратное считывание	Неповторяемое считывание	Фиктивные элементы
Read uncommitted (незавершенное считывание)	Возможно	Возможно	Возможно
Read committed (завершенное считывание)	Невозможно	Возможно	Возможно
Repeatable read (повторяемое считывание)	Невозможно	Невозможно	Возможно
Serializable (способность к упорядочению)	Невозможно	Невозможно	Невозможно

Как видите, по мере изменения уровня изоляции с «Read uncommitted» через «Read committed» и «Repeatable read» к наивысшему значению – «Serializable» – уменьшается количество возможных видов нежелательного поведения.

Уровень изоляции устанавливается при помощи команды SET TRANSACTION ISOLATION LEVEL, имеющей такой синтаксис:

```
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
```

По умолчанию будет установлен уровень READ COMMITTED.

Обратите внимание, что во время записи PostgreSQL не может обеспечить ни промежуточный уровень изоляции 1 «Repeatable read», ни входной – «Read uncommitted». Вообще говоря, уровень «Read uncommitted» означает возможность такого «плохого» поведения базы данных, что он предлагается лишь небольшим количеством СУБД, и редко встречаются приложения, которые оказываются настолько смелыми (или безрассудными), чтобы воспользоваться им.

Аналогично промежуточный уровень «Repeatable read» обеспечивает защиту только от фиктивных элементов, которые, как уже говорилось ранее, возникают чрезвычайно редко, поэтому отсутствие этого уровня изоляции не приводит ни к каким особым последствиям. Многие базы данных предлагают не весь набор возможностей, удачным компромиссным решением является обеспечение уровней «Read committed» и «Serializable».

Режимы явных и неявных транзакций (Auto Commit)

На протяжении всей этой главы для установки границ транзакций явно применялись операторы BEGIN WORK и COMMIT (или ROLLBACK) WORK. Но в начале книге, до знакомства с транзакциями, нам успешно удавалось производить изменения в базе данных без помощи команды BEGIN WORK.

По умолчанию PostgreSQL работает в режиме автоматической фиксации (auto commit mode), иногда также называемом **цепным режимом** (chained mode) или **режимом неявных транзакций** (implicit transaction mode), когда каждый оператор SQL, способный модифицировать данные, работает так, как если бы в его распоряжении имелась полная транзакция. Такой режим очень удобен для экспериментирования в командной строке, новые пользователи могут делать это, не обладая глубокими знаниями SQL, но он далеко не так хорош для реальных приложений, где мы стремимся получать доступ к транзакциям посредством явного оператора COMMIT или ROLLBACK.

В других SQL-серверах, реализующих другие режимы, обычно приходится давать явную команду изменения режима, например SET CHAINED в Sybase, или SET IMPLICIT_TRANSACTIONS в Microsoft SQL Server.

А в PostgreSQL надо просто ввести команду BEGIN WORK, и будет автоматически включен режим, в котором следующие за командой операторы включаются в транзакцию до тех пор, пока не вводится команда COMMIT или ROLLBACK.

Стандарт SQL рассматривает все SQL-операторы как происходящие в транзакции, при этом транзакция автоматически начинается при вводе первого оператора SQL и продолжается до тех пор, пока не встретится `COMMIT WORK` или `ROLLBACK WORK`. Поэтому в стандарте SQL не определена команда `BEGIN WORK`. Однако тот способ, которым выполняются транзакции в PostgreSQL, с явным вводом `BEGIN WORK`, также очень распространен.

Блокировки

Многие базы данных работают с транзакциями, в частности изолируют транзакции разных пользователей друг от друга, применяя блокировки для предотвращения доступа к данным со стороны других пользователей. Блокировки могут быть упрощенно разделены на два вида:

- Блокировка с **обеспечением совместного доступа** (shared lock), позволяющая другим пользователям читать, но не модифицировать данные.
- Блокировка **без совместного доступа**, или монопольная блокировка (exclusive lock), которая не разрешает другим пользователям даже читать данные.

Например, сервер блокирует строки, изменяемые транзакцией, до тех пор пока транзакция не закончится, тогда блокировки снимаются. Все это происходит автоматически, пользователи базы данных обычно даже не сознают, что происходит блокировка.

Подлинные механизмы блокирования и применяемые стратегии весьма сложны, в зависимости от обстоятельств могут применяться различные типы блокировок. Документация по PostgreSQL описывает семь разных видов блокировок. PostgreSQL также использует необычный механизм для изолирования транзакций, применяя поливариантную (multi-version) модель, которая уменьшает противоречия, возникающие между блокировками, и значительно улучшает производительность (по сравнению с другими схемами).

К счастью, пользователям базы данных, как правило, приходится беспокоиться о блокировках только в двух случаях, остерегаясь взаимных блокировок (и восстановления после них) и явного блокирования приложением.

Взаимные блокировки

Что произойдет, если два приложения одновременно попытаются изменить одни и те же данные? Чтобы воссоздать такую ситуацию, просто откроем две сессии `psql` и попытаемся изменить одну и ту же строку в каждом из них (табл. 9.8):

Таблица 9.8. Взаимные блокировки

Сессия 1	Сессия 2
`UPDATE row 14`	
	`UPDATE row 15`
`UPDATE row 15`	
	`UPDATE row 14`

В этой ситуации обе сессии блокируются, и каждая из них ждет, когда другая снимет блокировку.

Такое поведение объясняет, почему по умолчанию PostgreSQL устанавливает режим изоляции «Read committed». Он представляет собой компромисс между взаимной совместимостью, производительностью и минимизацией количества блокировок, с одной стороны, и непротиворечивостью и «идеальным» поведением – с другой (рис. 9.3):

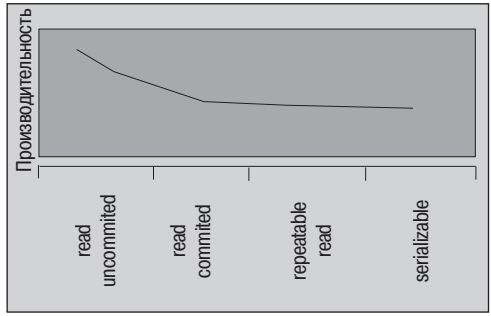


Рис. 9.3. Зависимость производительности базы данных от выбранного уровня изоляции

По мере того как база данных начинает вести себя менее «идеально», требуется все большее количество блокировок, взаимная совместимость пользователей уменьшается и общая производительность падает. Не очень удачный, но неизбежный компромисс.

В примере, приведенном выше, несмотря на то что сессия 2 была заблокирована до ожидания завершения сессии 1, ничего особенного не происходит, сессия 2 просто продолжается несколько дольше. Гораздо более серьезным является случай, когда обе сессии блокируют друг друга.

Попробуйте сами. Взаимные блокировки

Откроем две сессии `psql` и попробуем выполнить такую последовательность команд (табл. 9.9).

Таблица 9.9. Взаимные блокировки: практика

Сессия 1	Сессия 2
`BEGIN WORK`	
	`BEGIN WORK`
`UPDATE customer SET fname = 'D'` `WHERE customer_id = 15;`	
	`UPDATE customer SET fname = 'B'` `WHERE customer_id = 14;`
`UPDATE customer SET fname = 'Bill'` `WHERE customer_id = 14;`	
	`UPDATE customer SET fname = 'Dave'` `WHERE customer_id = 15;`

Обе сессии окажутся заблокированными, а через некоторое время в одной из них появится сообщение:

```
ERROR:  Deadlock detected.
        See the lock(l) manual page for a possible cause.
```

Другая сессия будет продолжена. Сессия, в которой появилось сообщение об ошибке, будет отменена, а сделанные изменения будут утеряны. Вторая сессия может продолжиться, в ней будет выполнен оператор `COMMIT WORK`, и в базе данных будут зафиксированы изменения.

PostgreSQL обнаруживает взаимную блокировку, если обе сессии блокируются, ожидая, пока выполнится вторая, и ни одна из них не двигается вперед.

Как это работает

Сессия 1 первой блокирует строку 15, затем возникает сессия 2 и блокирует строку 14. Теперь сессия 1 пытается заблокировать строку 14, но не может этого сделать, т. к. строка заблокирована сессией 2, а сессия 2 пытается изменить строку 15, но ей это не удается, потому что строка заблокирована сессией 1. По истечении небольшого периода имеющийся в PostgreSQL код обнаружения взаимных блокировок определяет, что произошла взаимная блокировка, и автоматически аннулирует транзакцию.

Невозможно заранее предугадать, какую именно сессию аннулирует PostgreSQL. СУБД попытается выбрать ту, которая, по ее мнению, «сделала меньше работы», но делает это «на глаз».

Приложения могут, и им следовало бы, предпринимать меры по избежанию возникновения взаимных блокировок. Простейший метод уже упоминался ранее: нужно делать транзакции максимально короткими. Чем меньше строк и таблиц задействованы в транзакции и чем меньше времени должна длиться блокировка, тем меньше шансов на возникновение конфликта.

Есть еще один, почти такой же простой метод: следует сделать так, чтобы приложение всегда обрабатывало таблицы и строки в одном и том же порядке. В нашем примере, если бы обе сессии пытались обновить строки в одном и том же порядке, то проблем бы не было, поскольку сессия 1 могла бы изменить обе свои строки и завершиться. Сессия 2 была бы ненадолго приостановлена и продолжена после завершения сессии 1. Можно также создать код, который бы повторял попытку после возникновения взаимной блокировки, но лучше постараться так спроектировать приложение, чтобы предупредить возникновение проблемы, чем писать код для повтора в случае неудачи.

Явные блокировки

Может случиться так, что автоматических блокировок, обеспечиваемых PostgreSQL, окажется недостаточно и потребуется явно заблокировать или несколько строк, или целую таблицу. Без необходимости применять явные блокировки не следует. Стандарт SQL даже не определяет способ блокирования целой таблицы, это расширение PostgreSQL.

Блокировать строки или таблицу можно лишь внутри транзакции. Как только транзакция завершена посредством `COMMIT` или `ROLLBACK`, все блокировки, сделанные во время транзакции, автоматически снимаются. Явное снятие блокировок во время транзакции невозможно по очень простой причине: снятие блокировки со строки, которая этой транзакцией была изменена, может позволить другому приложению изменить ее, что помешает выполнению оператора `ROLLBACK`, отменяющего начальное изменение.

Блокирование строк

Чаще всего бывает необходимо заранее заблокировать несколько строк, чтобы произвести в них изменения. Кстати, так можно избежать взаимных блокировок. Заблокировав все строки, которые необходимо будет изменить, мы гарантируем, что другие приложения не будут вмешиваться в изменения «на половине пути».

Чтобы заблокировать несколько строк, просто выполните оператор `SELECT`, добавив в него инструкцию `FOR UPDATE`:

```
SELECT 1 FROM item WHERE sell_price > 5.0 FOR UPDATE;
```

Учитывая, что мы находимся внутри транзакции, этот оператор заблокирует все строки таблицы `item`, в которых значение `sell_price` больше 5. В данном случае не требуется возвращать никакие строки, поэтому для минимизации возвращаемых данных просим вернуть просто 1.

Попробуйте сами. Блокирование строк

Предположим, что требуется заблокировать все строки таблицы customer, относящиеся к клиентам, проживающим в городе Nicetown, чтобы, например, изменить код города в номере телефона. Необходимо обеспечить доступ ко всем строкам, также потребуется некоторый программный код для последовательной обработки строк, с вычислением нового кода (если, например, междугородний код разделяется на несколько новых, зависящих от почтового индекса):

```
bpfinal=# BEGIN WORK;
BEGIN
bpfinal=# SELECT customer_id FROM customer WHERE town = 'Nicetown' FOR
bpfinal=# UPDATE;
 customer_id
-------------
           3
           6
(2 rows)

bpfinal=#
```

К этому моменту заблокированы две строки со значениями customer_id, равными 3 и 6, это можно проверить, попытавшись изменить их при помощи оператора UPDATE в другой сессии psql:

```
bpfinal=# BEGIN;
BEGIN
bpfinal=# UPDATE customer SET phone = '023 3376' WHERE customer_id = 2;
UPDATE 1
bpfinal=# UPDATE customer SET phone = '023 3267' WHERE customer_id = 3;
UPDATE 1

bpfinal=#
```

В данной точке вторая сессия блокируется и остается в таком состоянии до нажатия клавиш <Ctrl>+<C> для прерывания сессии или же до тех пор, пока не будет зафиксирована (посредством COMMIT) или отменена (при помощи ROLLBACK) первая сессия.

Как это работает

Первая сессия посредством SELECT … FOR UPDATE вызывает блокирование строк со значениями идентификаторов, равными 3 и 6. Другие сессии могут изменять другие строки таблицы customer, но не строки 3 и 6 (до тех пор, пока не завершится заблокировавшая их транзакция).

Блокирование таблиц

В PostgreSQL есть возможность блокировать таблицы, которой, однако, не рекомендуется пользоваться, поскольку в большинстве случаев это значительно ухудшает производительность.

Синтаксис операций выглядит следующим образом:

```
LOCK [ TABLE ] name
LOCK [ TABLE ] name IN [ ROW | ACCESS ] { SHARE | EXCLUSIVE } MODE
LOCK [ TABLE ] name IN SHARE ROW EXCLUSIVE MODE
```

Подробный рассказ о различных видах блокировок выходит за рамки данной книги, требуя глубокого понимания внутренней работы блокировок в базе данных. С деталями можно ознакомиться в оперативной справке.

Обычно приложения, которым необходимо заблокировать таблицу, поступают просто:

```
LOCK TABLE table-name
```

что является аналогом такого выражения:

```
LOCK TABLE table-name ACCESS EXCLUSIVE MODE.
```

Этим способом предотвращается любой доступ к таблице со стороны любого приложения. Такая мера может показаться драконовской, но в тех редких случаях, когда требуется блокировка на уровне таблицы, вероятно, необходим именно такой образ действий.

Резюме

В данной главе рассмотрены транзакции и блокировки. Показано, что применение транзакций полезно даже в однопользовательских базах данных, при этом команды SQL сгруппировываются вместе в единый элементарный блок, который или выполняется, или отменяется.

Рассказано о работе транзакций в многопользовательской среде. Представлены свойства АСИД для баз данных и пояснено, что означают свойства атомарности (atomicity), согласованности (consistency), изоляции (isolation) и долговечности (durability) в терминах базы данных.

Введены стандартные термины для возможных нежелательных явлений и определены различные уровни совместимости транзакций по отношению к данным видам нежелательных явлений. Приведено краткое обсуждение негативного влияния устранения нежелательных особенностей на производительность базы данных и необходимости нахождения компромисса между «идеальным» поведением и улучшением производительности.

Также рассмотрены простые способы, уменьшающие шансы возникновения взаимных блокировок, когда два (или более) приложения оказываются остановленными и каждое ожидает завершения работы другого.

В конце главы описаны явные блокировки, блокировка нескольких строк таблицы или даже целой таблицы на время выполнения транзакции.

Может быть, транзакции и блокировки – не самая интересная тема для обсуждения, но базовое понимание механизма их работы чрезвычайно важно для написания надежных приложений. Поэтому следует обеспечить не только их правильную работу, но и взаимодействие с базой данных, позволяющее минимизировать потери производительности и наилучшим образом эксплуатировать многопользовательские возможности PostgreSQL.

10

Хранимые процедуры и триггеры

В данной главе рассмотрены способы расширения возможностей PostgreSQL, изученных до настоящего момента. Бульшая часть материала, представленного в главе, относится только к PostgreSQL, хотя многие коммерческие СУБД, такие как Oracle, имеют подобные возможности.

Начнем с обсуждения операторов, которые PostgreSQL разрешает использовать внутри операторов `SELECT`, в том числе математических и операторов сравнения строк, позволяющих задавать более сложные условия в выражениях `WHERE`. Затем поговорим о том, как операторы в PostgreSQL реализуются в виде функций, и изучим несколько дополнительных функций, которые увеличивают мощь операторов `SELECT`.

PostgreSQL позволяет разработчику расширить функциональность сервера базы данных за счет написания функций на языке программирования C и загрузки их на сервер при запуске базы данных.

Расширение может быть просто отдельной дополнительной функцией, а может само быть таким же сложным, как целый язык программирования. Несколько таких расширений, известных как загружаемые процедурные языки, включены в стандартный набор поставки PostgreSQL. Эти языки позволяют создавать свои собственные функции быстрее и проще, чем если бы пришлось писать их на C.

Один из загружаемых языков, PL/pgSQL, будет кратко представлен в данной главе. Есть и другие, такие как PL/Tcl и PL/Perl, позволяющие создавать расширения PostgreSQL на языках программирования Tcl и Perl соответственно.

Функции, созданные как расширения, выполняются сервером, а не клиентским приложением и хранятся внутри самой базы данных. Они известны как «хранимые процедуры».

В данной главе приведены примеры хранимых процедур, созданных посредством PL/pgSQL, который доступен только в PostgreSQL, но во многих СУБД поддерживаются подобные языки. Например, в Oracle есть PL/SQL, а в Sybase – Transact-SQL.

В определенных условиях (касающихся состояния базы данных) сервер PostgreSQL может автоматически запускать хранимые процедуры. Например, если предпринимается попытка удалить из таблицы строку, то хранимая процедура может быть выполнена для обеспечения ссылочной целостности (удалит соответствующие строки в других таблицах), или же она может помешать удалению. Такие автономные операции называются триггерами, и в этой главе мы увидим их в действии.

В данной главе описаны:

- Операторы
- Функции
- Процедурные языки
- PL/pgSQL
- Хранимые процедуры
- SQL-функции
- Триггеры

Операторы

В главе 8 и далее внутри выражений оператора `SELECT` уже использовались некоторые простые операторы. Например, можно при помощи оператора числового сравнения ограничить выборку строками, удовлетворяющими некоторому условию (выбрать изделия, себестоимость которых превышает 4 доллара):

```
bpfinal=# SELECT * FROM item WHERE cost_price > 4;
 item_id |  description  | cost_price | sell_price
---------+---------------+------------+------------
       1 | Wood Puzzle   |      15.23 |      21.95
       2 | Rubik Cube    |       7.45 |      11.49
       5 | Picture Frame |       7.54 |       9.95
       6 | Fan Small     |       9.23 |      15.75
       7 | Fan Large     |      13.36 |      19.95
      11 | Speakers      |      19.73 |      25.32
(6 rows)

bpfinal=#
```

Оператор > задает отношение между атрибутом `cost_price` и указанным числом.

Давайте пойдем дальше и зададим более сложные условия, введя дополнительные выражения и операторы:

```
bpfinal=# SELECT * FROM item WHERE (sell_price*100)%100 = 99;
item_id | description | cost_price | sell_price
--------+-------------+------------+------------
       4 | Tissues     |       2.11 |       3.99
(1 row)

bpfinal=#
```

Использован оператор умножения с оператором взятия остатка от деления, таким образом выделены изделия, значение себестоимости которых оканчивается 99 центами.

Рассмотрим другой оператор. Он выполняет проверку соответствия регулярному выражению без учета регистра, определяя, какие изделия имеют описание, начинающееся с «p» или «r» и заканчивающееся буквой «e»:

```
bpfinal=# SELECT * FROM item WHERE description ~* '^[PR].*E$';
 item_id |  description  | cost_price | sell_price
---------+---------------+------------+------------
       2 | Rubik Cube    |       7.45 |      11.49
       5 | Picture Frame |       7.54 |       9.95
(2 rows)

bpfinal=#
```

Как видите, PostgreSQL поддерживает множество операторов. Оказывается, если принимать в расчет разновидности одного и того же оператора (то есть отличать сравнение целых чисел от сравнения чисел с плавающей точкой и сравнения строк), то получится, что нам доступны почти 600 операторов.

Приоритет и ассоциативность операторов

Многие операторы PostgreSQL выглядят и действуют во многом похоже на обычные арифметические операторы, присутствующие в большинстве языков программирования. У операторов существует жестко запрограммированный в синтаксическом анализаторе приоритет, который определяет порядок их выполнения в сложных выражениях. Как обычно, можно изменить приоритет при помощи скобок.

Обратите внимание, что PostgreSQL разрешает применять операторы (и, как будет показано позже, функции) вне выражений `WHERE` операторов `SELECT`:

```
bpfinal=# SELECT 1+2*3;
 ?column?
------------
```

```
        7
(1 row)
bpfinal=# SELECT (1+2)*3 AS answer;
 answer
----------
      9
(1 row)

bpfinal=#
```

Как видите, результат вычисления выражения `1+2*3` представлен как `7`, он выведен как неизвестный столбец, имеющий по умолчанию имя `?column?`. Во втором примере приоритет операторов изменен, и результату дано название `answer`.

Несмотря на то что некоторые операторы ведут себя именно так, как предполагают пользователи, имеющие опыт программирования на C или другом языке программирования, есть случаи, в которых приоритет оказывается совсем не очевидным. Как и в C, булевы операторы имеют более низкий приоритет, чем арифметические, поэтому для обеспечения требуемого порядка выполнения операций часто требуется применение скобок. Если вы не знаете точно, приоритет какого оператора выше, задайте порядок выполнения явно при помощи скобок.

Операторам PostgreSQL также присуща ассоциативность слева или справа, определяющая порядок выполнения операторов с одинаковым приоритетом. Арифметические операторы, такие как сложение и вычитание, ассоциативны слева, поэтому выражение 1+2-3 вычисляется так, как если бы оно было записано в форме (1+2)-3. Другие операторы, например логическое сравнение, ассоциативны справа, поэтому выражение x = y = z оценивается как x = (y = z).

Наиболее употребительные операторы PostgreSQL, упорядоченные по убыванию лексических приоритетов, представлены в табл. 10.1:

Таблица 10.1. Приоритетность основных операторов PostgreSQL

Оператор	Ассоциативность	Значение
`UNION`	Слева	Выражение SQL оператора `SELECT`
`::`		Приведение типа (синоним `CAST`)
`[]`	Слева	Выбранный элемент массива
`.`	Слева	Выбранный атрибут (столбец)
`-`	Справа	Унарный минус (целое отрицание)
`^`	Слева	Возведение в степень
`* / %`	Слева	Операторы умножения и деления
`+ -`	Слева	Аддитивные операторы
`IS`		Проверка (на `TRUE`, `FALSE` и `NULL`)

Оператор	Ассоциативность	Значение
ISNULL		Проверка (на NULL)
NOTNULL		Проверка (на не-NULL)
OR	Слева	Логическое сложение
IN		Проверка на принадлежность множеству
BETWEEN		Проверка на вхождение в диапазон
LIKE		Проверка на соответствие шаблону
<>		Проверка на неравенство
=	Справа	Проверка на равенство
NOT	Справа	Логическое отрицание
AND	Слева	Логическое умножение
Все остальные операторы		Все определенные пользователями и встроенные операторы, не перечисленные выше, имеют одинаковый приоритет

PostgreSQL поддерживает операторы вычисления натуральных логарифмов и антилогарифмов (`:` ***и*** `;` ***), но эти операторы не рекомендованы к использованию и могут быть не включены в следующие версии PostgreSQL. Используйте вместо них функции*** `ln()` ***и*** `exp()`.

Арифметические операторы

В PostgreSQL доступно множество арифметических операторов. Перечень основных арифметических операторов приведен в табл. 10.2. Все они являются ассоциативными слева и имеют одинаковый приоритет:

Таблица 10.2. Основные арифметические операторы

Оператор	Пример	Значение
+	2+3 равно 5	Сложение
-	3-2 равно 1	Вычитание
*	2*3 равно 6	Умножение
/	3/2 равно 1 3/2.0 равно 1.5 3/2::float8 равно 1.5	Деление
%	22 % 7 равно 1	Остаток (деление по модулю)
^	4^3 равно 64	Возведение в степень
&	14 & 23 равно 6	Поразрядное И
|	14 | 23 равно 31	Поразрядное ИЛИ
#	14 # 23 равно 25	Поразрядное исключающее ИЛИ
>>	128 >> 4 равно 8	Сдвиг вправо
<<	1 << 4 равно 16	Сдвиг влево

Существуют также унарные арифметические операторы, они приведены в табл. 10.3:

Таблица 10.3. Унарные арифметические операторы

Оператор	Пример	Значение
%	%2.3 равно 2	Округление в меньшую сторону
!	4! равно 24	Факториал
!!	!!4 равно 24	Факториал как префиксный оператор
@	@-2 равно 2	Значение по модулю
|/	|/64 равно 8	Квадратный корень
||/	||/64 равно 4	Кубический корень
~	~15 равно −16	Побитовая инверсия

В целом арифметические операторы «все делают правильно». PostgreSQL использует ту версию оператора, которая соответствует заданному аргументу. Так, при делении одного целого числа на другое результат будет целым числом. При делении одного числа с плавающей точкой на другое результат будет числом с плавающей точкой. Как показано в примере таблицы, чтобы изменить тип результата на число с плавающей точкой, один из аргументов необходимо привести к типу числа с плавающей точкой.

Сравнения и операции над строками

PostgreSQL предоставляет обычный набор операторов сравнения (табл. 10.4), таких как «меньше чем» и «больше чем». Такие операторы могут быть применены к большей части типов, поддерживаемых PostgreSQL, например можно применять оператор «больше чем» для проверки алфавитного порядка следования строк, а также для сравнения числовых значений.

Результатом сравнения может быть или TRUE, или FALSE. В psql они отображаются как t и f:

Таблица 10.4. Операторы сравнения

Оператор	Пример	Значение
<	2 < 3 'axy' < 'azz'	Меньше
<=	2 <= 3	Меньше или равно
<>	2 <> 3	Не равно
!=	2 != 3	
=	3 = 1+2	Равно
>	3 > 2	Больше

Оператор	Пример	Значение
>=	3 >= 2	Больше или равно
IN	3 in (1,2,3)	Принадлежит множеству
!!=	4 !!= (2,3)	Не принадлежит множеству (синоним NOT IN)

Для операций над строками в PostgreSQL существует специальный набор операторов. В него входят операторы объединения строк (конкатенации) и операторы проверки соответствия по различным правилам (табл. 10.5):

Таблица 10.5. Строковые операторы

Оператор	Пример	Значение
||	'abc' || 'def' равно 'abcdef'	Конкатенация строк
~~	'xyzzy' ~~ '%zz%'	Синоним LIKE
!~~	'xyzzy' !~~ '%aa%'	Синоним NOT LIKE
~	'xyzzy' ~ 'y.*y'	Соответствие подстроки регулярному выражению. Использование символа ^ в начале регулярного выражения привязывает поиск соответствия к началу строки, а заключительный знак $ – к концу, возможно и применение обоих символов
~*	'xyzzy' ~* '^X.*Y$'	Соответствие регулярному выражению без учета регистра
!~	'xyzzy' !~ 'aa'	Несоответствие регулярному выражению (обратно по отношению к ~)
!~*	'xyzzy' !~* 'AA'	Несоответствие регулярному выражению без учета регистра (обратно по отношению к ~*)

При проверке соответствия регулярному выражению строка сравнивается с выражением, подобным тем, которые используются в UNIX-утилите grep или в операторах проверки соответствия Perl.

Другие операторы

PostgreSQL поддерживает и массу дополнительных операторов для сравнения и обработки присущих только PostgreSQL типов данных, таких как точки, круги, временные интервалы и IP-адреса. Подробная информация приведена в руководстве пользователя по PostgreSQL.

Документация по PostgreSQL доступна в формате HTML-страниц, которые могут быть просмотрены любым броузером. Выберите file://usr/local/pgsql/doc/html.

Все операторы перечислены в таблице pg_operator базы данных, а в psql можно вывести список всех операторов и функций при помощи внутренних команд \do и \df.

Функции

В PostgreSQL реализовано огромное количество встроенных функций, которые могут использоваться в выражениях для SELECT.

В перечень входят:

- Функциональные эквиваленты уже рассмотренных операторов
- Дополнительные математические функции
- Дополнительные функции обработки строк
- Функции обработки дат и времен
- Функции форматирования текста
- Функции для геометрических типов PostgreSQL, таких как точки и круги
- Функции для IP-адресов

Встроенные функции (а на самом деле и определенные пользователем функции) записаны в системную таблицу базы данных PostgreSQL, pg_proc. В PostgreSQL версии 7.1 и выше в данной таблице содержится более 1100 записей.

Посредством команды \df в psql можно вывести все функции и их аргументы. Пояснения для какой-то конкретной функции или группы функций доступны по команде \dd.

Нельзя даже мечтать о том, чтобы рассказать обо всех функциях в одной главе, но, по крайней мере ненадолго, остановимся на самых полезных. Некоторые стандартные функции уже встречались нам в главах 7 и 8. За более полной информацией обращайтесь к руководству пользователя PostgreSQL. Можно также просмотреть регрессивные тесты, распространяемые с исходными кодами PostgreSQL.

Многие встроенные функции служат эквивалентами для всех математических и логических операторов. В качестве примера укажем int4shl и float8mul, аналоги оператора сдвига влево (<<) для целых чисел и оператора умножения (*) для чисел с плавающей точкой, соответственно.

Список дополнительных математических функций представлен в табл. 10.6. Все эти функции применяются к числам с плавающей точкой и возвращают число с плавающей точкой, если не заданы другие условия.

Таблица 10.6. Дополнительные математические функции

Функция	Значение
abs(x)	Абсолютное значение
degrees(r)	Перевод из радиан в градусы
radians(d)	Перевод из градусов в радианы
exp(x)	Натуральный антилогарифм, возведение *e* в степень
ln(x)	Натуральный логарифм
log(x)	Логарифм по основанию 10
log(b,x)	Логарифм по заданному основанию, b
mod(x,y)	Остаток от деления x на y (имеется версия для целых чисел)
pi()	Возвращает число π
pow(x,y)	Возведение x в степень y
random()	Возвращает случайное число из промежутка от 0.0 до 1.0
round(x)	Округляет до ближайшего целого
round(x,d)	Округляет с точностью до указанного количества десятичных разрядов, d
trunc(x)	Округляет до целого в сторону нуля
Trunc(x,d)	Округляет до заданного количества разрядов, d
ceil(x)	Возвращает минимальное целое, превышающее заданный аргумент
floor(x)	Возвращает максимальное целое, не превышающее заданный аргумент
sqrt(x)	Квадратный корень
cbrt(x)	Кубический корень
float8(i)	Преобразует целое число в эквивалентное число с плавающей точкой float8
float4(i)	Преобразует целое число в эквивалентное число с плавающей точкой float4
int4(x)	Возвращает целое, при необходимости округляет

Поддерживаются в PostgreSQL и тригонометрические функции (табл. 10.7, все аргументы и результаты указываются в радианах):

Таблица 10.7. Тригонометрические функции

Функция	Значение
sin	Синус
cos	Косинус
tan	Тангенс
cot	Котангенс

Таблица 10.7. (продолжение)

Функция	Значение
asin	Арксинус
acos	Арккосинус
atan	Арктангенс
atan2	Арктангенс с двумя аргументами. Если заданы a, b, то вычисляется atan(b/a)

PostgreSQL включает в себя стандартные строковые функции SQL (табл. 10.8) с их оригинальным синтаксисом. Для таких функций строка должна иметь тип char, varchar или text:

Таблица 10.8. Стандартные строковые функции SQL

Функция	Значение
char_length(s) character_length(s)	Длина строки
octet_length(s)	Количество памяти, занятое строкой, в байтах
lower(s)	Преобразует строку в нижний регистр
upper(s)	Преобразует строку в верхний регистр
position(s1, s2)	Позиция, на которой s1 появляется в s2
substring(s from n for m)	Выделяет подстроку длиной m, начинающуюся с позиции n
trim([leading | trailing | both] [s1] from s2)	Удаляет символы s1 из строки s2 или с начала, или из конца, или с обеих сторон. По умолчанию удаляет пробелы, если не задано s1

Стандартные возможности обработки строк в PostgreSQL дополнены за счет своих собственных дополнительных функций. Обращайтесь за информацией к руководству пользователя.

Стоит также упомянуть важную форматирующую функцию, to_char. Она играет в PostgreSQL ту же роль, что printf в C, то есть отвечает за любые форматирования значений для печати или отображения на терминале. Она отформатирует дату и время согласно шаблону для дат, а числовое значение может отформатировать множеством различных способов, в том числе и как римскую цифру. Подробное описание приведено в руководстве пользователя.

Процедурные языки

Как было сказано в начале данной главы, в PostgreSQL есть возможность определять свои собственные функции, предназначенные для работы в пределах базы данных. Это свойство может быть полезным,

если какой-то отдельный запрос или какое-то вычисление требуется применять еще во многих других местах.

Для создания новой функции предназначен SQL-оператор CREATE FUNCTION, синтаксис которого представлен ниже:

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    AS definition
    LANGUAGE 'langname'
```

Существует и другая форма оператора* CREATE FUNCTION*, которая делает возможной включение в состав сервера PostgreSQL откомпилированного объектного кода, обычно созданного из исходного кода, написанного на C. Расширение возможностей сервера за счет написания функций на C выходит за рамки изложения этой главы.

Одна из простейших функций, просто увеличивающая свой единственный аргумент, может быть написана так:

```
CREATE FUNCTION add_one(int4) RETURNS int4 AS '
    BEGIN
        RETURN $1 + 1;
    END;
' LANGUAGE 'plpgsql';
```

Определение функции задается в виде одной символьной строки, которая может занимать несколько строчек и может быть написана на любом языке, поддерживаемом PostgreSQL в качестве загружаемого процедурного языка. В данном случае использовался PL/pgSQL, о чем свидетельствует значение plpgsql, указанное в выражении LANGUAGE.

Чтобы иметь возможность обрабатывать процедурный язык, необходимо сначала расширить PostgreSQL за счет присоединения функции-обработчика, обычно написанной на C. Обработчик для PL/pgSQL включен в пакет поставки в виде разделяемой (shared) библиотеки.

Когда функция создана, ее определение сохраняется в базе данных. При первом вызове функции обработчик компилирует ее в исполняемую форму и выполняет. То есть уведомление об ошибке в функции можно не получить до тех пор, пока функция не будет использована.

Прежде чем создавать собственные функции на любом из выбранных загружаемых языков, необходимо сделать так, чтобы PostgreSQL поддерживала этот язык. Поработаем над этим.

Основы PL/pgSQL

В данной главе для создания образцов хранимых процедур в качестве загружаемого процедурного языка рассматривается PL/pgSQL. В стандартной конфигурации PostgreSQL обработчик для PL/pgSQL

входит в состав библиотеки plpgsql.so, расположенной в каталоге /usr/local/pgsql/lib для UNIX и Linux, или библиотеки plpgsql.dll из каталога /usr/lib, если применяется Cygwin под Microsoft Windows.

Каждая база данных PostgreSQL является независимой по отношению к поддерживаемым языкам, отчасти из соображений безопасности. Можно создать функции, которые будут случайно или же злонамеренно поглощать все ресурсы CPU, например, зациклившись. Это может послужить основой для атаки типа «отказ в обслуживании». Поэтому по умолчанию базы данных PostgreSQL не поддерживают процедурные языки. Чтобы использовать PL/pgSQL, необходимо самостоятельно инсталлировать обработчик.

Администратор базы данных может добавить языки в базу template1, тогда во всех новых базах данных эти языки будут присутствовать по умолчанию.

Чтобы установить PL/pgSQL для базы данных bpsimple, можно применить в psql команду CREATE LANGUAGE и загрузить разделяемую библиотеку, явно создав функцию-обработчик. Это достаточно сложно для того, чтобы существовал соответствующий вспомогательный сценарий, и он действительно присутствует в дистрибутиве PostgreSQL. Нужная нам команда называется createlang:

```
createlang [options] [langname] dbname
```

Ее параметры представлены в табл. 10.9:

Таблица 10.9. Параметры команды createlang

Параметр	Значение
-h, --host=HOSTNAME	Имя сервера базы данных
-p, --port=PORT	Порт сервера базы данных
-U, --username=USERNAME	Имя пользователя для осуществления соединения
-W, --password	Приглашение на ввод пароля
-d, --dbname=DBNAME	База данных, в которую будет устанавливаться язык
-L, --pglib=DIRECTORY	Искать файл интерпретатора языка в DIRECTORY
-l, --list	Показывает список языков, установленных на текущий момент

Если название языка langname не указано, то появится приглашение ввести его. Указание имени базы данных dbname обязательно. Обычные пользователи не имеют прав на добавление языков в базу данных, поэтому соединение будем осуществлять как привилегированный пользователь с именем postgres:

```
$ createlang -U postgres plpgsql bpsimple -L/usr/local/pgsql/lib
```

Можно проверить, присутствует ли язык в базе, выведя список языков при помощи `createlang` или сделав выборку из системной таблицы `pg_language` с помощью `psql` и `pgAdmin`:

```
$ createlang -l bpsimple
     Procedural languages
  Name   | Trusted? | Compiler
---------+----------+----------
 plpgsql | t        | PL/pgSQL
(1 row)

$ psql -d bpsimple

bpfinal=# SELECT * FROM pg_language;
 lanname  | lanispl | lanpltrusted | lanplcallfoid | lancompiler
----------+---------+--------------+---------------+-------------
 internal | f       | f            |             0 | n/a
 C        | f       | f            |             0 | /bin/cc
 sql      | f       | f            |             0 | postgres
 plpgsql  | t       | t            |         21571 | PL/pgSQL
(4 rows)

bpfinal=#
```

Привилегированные пользователи могут удалить поддержку языков, выполнив команду `DROP LANGUAGE` в `psql`:

```
bpfinal=# DROP language 'plpgsql';
DROP

bpfinal=#
```

Попробуйте сами. Первая хранимая процедура

Теперь мы готовы приступить к работе с хранимыми процедурами PL/pgSQL, то есть к написанию собственных функций. Но сначала проверим, все ли работает, выполнив функцию `add_one`, описанную ранее:

```
bpfinal=# CREATE function add_one (int4) RETURNS int4 as '
bpfinal'# BEGIN return $1 + 1; end;' language 'plpgsql';
CREATE
bpfinal=# SELECT add_one(2) AS answer;
 answer
--------
      3
(1 row)

bpfinal=#
```

Как это работает

Команда CREATE FUNCTION сохраняет определение функции add_one, написанной на PL/pgSQL, в базе данных. Она выполняется при оценке выражения SELECT. Обратите внимание, что PL/pgSQL не воспринимает ни регистр ключевых слов, таких как BEGIN, ни форматирование. Функцию можно было определить и следующим образом:

```
bpfinal=# CREATE function
bpfinal-# add_one(int4) RETURNS int4
bpfinal-# AS '
bpfinal'# BEGIN
bpfinal'#   RETURN $1 + 1;
bpfinal'# END;
bpfinal'# '
bpfinal-# LANGUAGE 'plpgsql';
CREATE

bpfinal=#
```

Перегрузка функций

PostgreSQL рассматривает функции как различные, если они имеют разные имена, разное количество параметров или если их параметры принадлежат к разным типам данных. При желании можно создать функции add_one, которые имеют дело с разными типами данных. Посмотрим, что произойдет, если использовать написанную ранее функцию add_one для значения с плавающей точкой:

```
bpfinal=# SELECT add_one(3.1);
 add_one
---------
       4
(1 row)

bpfinal=#
```

Как видите, PostgreSQL выполнила нашу функцию add_one, которая, как указано типом параметра (int4), принимает целый параметр. Введенное значение (3.1) было приведено к целому типу, а именно – округлено до 3, и это значение затем было увеличено.

Если потребуется создать функцию приращения для чисел с плавающей точкой, просто создадим другое определение add_one:

```
bpfinal=# CREATE function
bpfinal-# add_one(float8) RETURNS float8
bpfinal-# AS '
bpfinal'# BEGIN
bpfinal'#   RETURN $1 + 1;
bpfinal'# END;
bpfinal'# '
```

```
bpfinal-# LANGUAGE 'plpgsql';
CREATE
bpfinal=# SELECT add_one(3.1);
 add_one
----------
     4.1
(1 row)

bpfinal=#
```

Теперь получен ожидаемый результат, т. к. PostgreSQL выполнила правильную версию функции `add_one`. Такое поведение, называемое **перегрузкой функций**, может быть очень полезным, но может и вносить путаницу. Чтобы отличать функции, следует ссылаться на них, указывая их параметры. В данном случае есть две функции, на которые можно ссылаться как на `add_one(int4)` и `add_one(float8)`.

Более удобным способом создания функций является редактирование файлов сценария, содержащих определения функций, и запуск утилиты `psql` ***с ключом*** `\i` ***для их чтения.***

Листинг функций

Для того чтобы просмотреть исходный текст функций после их загрузки в базу данных, можно осуществить запрос в таблице, отведенной для хранимых процедур. Это таблица `pg_proc`:

```
bpfinal=# SELECT prosrc FROM pg_proc WHERE proname = 'add_one';
            prosrc
------------------------------

begin
        return $1 + 1;
end;

begin
        return $1 + 1;
end;

(2 rows)

bpfinal=#
```

Удаление функций

Функции можно удалить из базы данных при помощи `DROP FUNCTION`. Следует указать правильную версию для перегруженных функций, а если необходимо удалить функцию целиком, то должны быть удалены все версии:

```
bpfinal=# DROP function add_one(int4);
DROP
```

```
bpfinal=# DROP function add_one(float8);
DROP

bpfinal=#
```

Применение кавычек

В случае применения PL/pgSQL для хранимых процедур возникает одна небольшая сложность, связанная с кавычками. Полное определение функции передается команде CREATE FUNCTION в виде одной символьной строки, заключенной в кавычки. Это означает, что если внутри определения функции есть одинарные кавычки, их следует экранировать. Для этого перед каждой кавычкой внутри строки ставится еще одна кавычка. В тех случаях, когда это необходимо, в примерах данной главы кавычки экранируются.

Структура хранимой процедуры

Итак, пробная хранимая процедура создана, выполнена и удалена, и можно переходить к более тщательному анализу конструкции хранимой процедуры PL/pgSQL.

PL/pgSQL – это язык, имеющий блочную структуру, как Паскаль и C, с объявлением переменных и областями видимости. Для каждого блока существует необязательная метка, он может содержать несколько объявлений переменных и включать в себя операторы, образующие блок между ключевыми словами BEGIN и END. Синтаксис блока выглядит следующим образом:

```
[<<label>>]
[DECLARE declarations]
BEGIN
    statements
END;
```

PL/pgSQL не различает регистры. Все ключевые слова и названия переменных могут быть введены в любом регистре.

Функция PL/pgSQL определяется при помощи оператора CREATE FUNCTION, при этом частью определения является блок, заключенный в одинарные кавычки:

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    AS 'block definition'
    LANGUAGE 'plpgsql';
```

Аргументы функций

Функция PL/pgSQL может принимать ноль и более параметров, при этом типы параметров указаны в скобках после имени функции. Это встроенные типы PostgreSQL, такие как `int4` или `float8`. Все хранимые процедуры должны возвращать значение, а тип возвращаемого значения задан в выражении `RETURNS` определения функции.

В теле функции ее параметры обозначаются как `$1`, `$2` и т. д., в том порядке, в котором они определены. Дальше будет показано, что можно задать имена параметров с помощью объявления `ALIAS`.

Приведем пример простой хранимой процедуры, вычисляющей среднее геометрическое (с плавающей точкой) для двух целых чисел:

```
-- geom_avg
-- get a geometric average of two integers
create function geom_avg(int4, int4) returns float8 as '
begin
    return sqrt($1 * $2::float8);
end;
' language 'plpgsql';
```

Одно из целых значений необходимо привести к типу с плавающей точкой, чтобы результат умножения был передан функции `sqrt` в виде числа с плавающей точкой. Если этого не сделать, будет выдано сообщение об ошибке, информирующее о том, что функция `sqrt(int4)` не существует.

Комментарии

Вы могли заметить (в рассмотренных ранее примерах), что PL/pgSQL разрешает включать в определение функции комментарии. Различают два вида комментариев: однострочные и блочные.

Стандартный для SQL однострочный комментарий предваряется двумя дефисами (--). Все, начиная от двойного дефиса и заканчивая концом строки, игнорируется:

```
-- Это однострочный комментарий
create   -- комментарии могут появляться в
function -- любом месте и распространяться до конца строки
```

Блочные комментарии предназначены для ввода в качестве комментариев больших блоков текста или для временного удаления ненужных частей программы. Их синтаксис аналогичен C и C++, блоки комментариев начинаются символами `/*` и завершаются `*/`:

```
/*
      Это блочный комментарий, предназначенный для описания
      применения и поведения следующей функции
```

```
*/
create function blah() returns integer as '
begin
      /* закомментировать вызов функции
      func();
      */
      return 1;

end;
' language 'plpgsql';
```

Вложение блочных комментариев не разрешено, но при необходимости можно применять однострочные комментарии для предотвращения специальной интерпретации символов-ограничителей блочного комментария.

Объявления

Функции PL/pgSQL могут объявлять локальные переменные, используемые в их пределах. Каждая переменная принадлежит к какому-то типу, который может быть встроенным типом PostgreSQL, типом, определенным пользователем, или же типом, соответствующим строке таблицы.

Объявления переменных для функции записываются в раздел DECLARE определения функции или в блок DECLARE внутри функции.

Как это обычно бывает в блочных языках типа C и C++, переменные, определенные для блока, видны только в этом блоке или в блоках, входящих в данный. Переменная, объявленная во внутреннем блоке с тем же именем, что и переменная вне этого блока, скрывает внешнюю переменную:

```
DECLARE
      n1 integer;
      n2 integer;
BEGIN
      -- внутри можно использовать n1 и n2
      n2 := 1;
      DECLARE
            n2 integer; -- скрывает предыдущую n2
            n3 integer;
      BEGIN
            -- внутри можно использовать n1, n2 и n3
            n2 := 2;
      END;
      -- n3 здесь уже недоступна
      -- n2 здесь все еще имеет значение 1
END;
```

Все переменные, адресуемые в функции, должны быть объявлены до этого. Исключением являются только переменные управления циклом, о которых мы поговорим позже. Имена переменных не могут совпадать с зарезервированными словами PL/pgSQL (табл. 10.10):

Таблица 10.10. Зарезервированные слова PL/pgSQL

alias	begin	bpchar
Char	constant	debug
declare	Default	diagnostics
else	end	Exception
execute	exit	for
From	Get	if
in	Into	loop
not	notice	Null
perform	processed	raise
Record	rename	result
return	Reverse	select
then	to	Type
varchar	when	while

Существует несколько способов объявления переменной, выбор зависит от того, как предполагается конкретную переменную использовать.

ALIAS

Это простейшее объявление, оно дает имя параметру функции. Благодаря этому программа становится чуть более понятной и более устойчивой к изменениям номеров и порядка следования параметров. Объявление ALIAS имеет такой синтаксис:

```
name ALIAS FOR $n;
```

Становится доступной новая переменная с именем name, она действует как еще одно имя для указанного параметра. Например, можно было написать функцию geom_avg так:

```
create function geom_avg(integer, integer) returns float8 as '
declare
    first alias for $1;
    second alias for $2;
begin
    return sqrt(first * second::float8);
end;
' language 'plpgsql';
```

RENAME

Переменные можно переименовывать при помощи объявления RENAME. Как вы узнаете дальше, такая возможность полезна для переменных, находящихся внутри триггерной функции, но в большинстве случаев лучше не использовать ее, т. к. это затрудняет чтение кода. Синтаксис объявления RENAME таков:

```
RENAME original TO new;
```

Объявление простой переменной

Простая переменная объявляется посредством указания ее имени, типа и (необязательно) начального значения. Вот синтаксис объявления:

```
name [CONSTANT] type [NOT NULL] [:= value];
```

Модификатор CONSTANT показывает, что переменная не может быть изменена. Соответственно, в такое объявление обязательно должно входить начальное значение.

Выражение NOT NULL сообщает PostgreSQL о необходимости сгенерировать ошибку времени выполнения в случае, если переменная когда-либо получит значение NULL.

Начальное значение не обязательно должно быть константой, оно оценивается и присваивается при каждом вызове функции. Например, если задать начальное значение now для переменной timestamp, то переменная будет получать значение текущего времени при каждом выполнении, а не при компиляции.

Тип (type) может быть встроенным типом PostgreSQL. Другими словами, можно объявлять переменные тех же типов данных или той же структуры, что и другие элементы базы данных. Преимущество указания типа переменных в такой неявной форме состоит в том, что при внесении изменений в базу данных код хранимой процедуры остается корректным. Представим синтаксис:

```
builtintype
variable%TYPE
table.column%TYPE
```

Приведем несколько примеров объявления переменных:

```
n integer := 1;
mypi constant float8 := pi();
pizza_pi mypi%TYPE;
mydesc item.description%type := ''extra large size pizza'';
```

Объявлена переменная mydesc, соответствующая столбцу описания изделия таблицы item учебной базы данных. Если в настоящий момент это, например, char(64), а в дальнейшем превратится в char(80), то код,

в котором используется mydesc, сохранит работоспособность и PostgreSQL создаст корректный тип для переменной. Заметьте, что строка инициализации заключена в две пары кавычек, поэтому данное объявление появится внутри строки, заключенной в одинарные кавычки (определение функции).

Объявление составной переменной

В хранимых процедурах можно объявлять и использовать составные переменные. Они соответствуют целым строкам некоторой таблицы.

ROWTYPE

Для объявления составной переменной обратимся к синтаксису объявления ROWTYPE:

```
name table%ROWTYPE;
```

В результате такого объявления появится переменная, имеющая поля, по одному для каждого столбца таблицы, на базе которой она создана. Рассмотрим пример:

```
contact customer%ROWTYPE;
```

Создается переменная с именем contact, поля которой соответствуют столбцам таблицы customer. Для доступа к полю применяется обозначение variable.field. Пример:

```
DECLARE
    contact customer%ROWTYPE;
    address text;
BEGIN
    contact.zipcode := ''XY1 6ZZ'';
    contact.fname := NULL;
    address := contact.addressline || contact.town;
END;
```

RECORD

Существует еще один составной тип данных – RECORD. Этот тип во многом похож на ROWTYPE, но при его определении нет необходимости ссылаться на конкретную таблицу. В записи будут содержаться поля, соответствующие тому, что будет присвоено записи во время выполнения программы. Применение записей бывает удобным в программах, которые должны работать как триггеры, вызываемые из разных таблиц. Они также могут применяться обычным способом для хранения результатов операторов SELECT. Объявить запись очень просто:

```
name RECORD;
```

Более подробно о записях будет рассказано при изучении триггеров и присваиваний посредством выборок.

Присваивания

Новые значения переменных PL/pgSQL задаются операторами присваивания. Синтаксис операции таков:

```
reference := expression;
```

reference — это имя переменной или поля составной переменной, принадлежащей к типу rowtype или record. Выражение (expression) может быть константой, другой переменной или ссылкой на поле сложного выражения, составленного из операторов, приведений типов и вызовов функций. Приведем примеры присваиваний:

```
n1: = 23;
long_variable_names_are_OK := (n1 + 45)/2;
f2 := add_one(n1)::float8 * sqrt(2.0);

/* Для составных типов значение каждый раз может быть присвоено только одному
полю с помощью ссылки на это конкретное поле: */

contact.zipcode := ''AB12 3CD'';
```

Оператор SELECT INTO

Другой механизм присваивания реализуется с помощью расширения оператора SELECT. Оператор SELECT INTO позволяет присвоить значение переменной, списку переменных, переменной типа строки или записи. Синтаксис является расширением обычного SQL-оператора SELECT:

```
SELECT expressions INTO target [FROM ...];
```

Вот несколько простых примеров использования SELECT вместо оператора присваивания:

```
SELECT sqrt(2.0) INTO sqrt2;
SELECT add_one(n1) INTO n1;
SELECT 1,2,3,4 INTO n1, n2, n3, n4;
SELECT ''Mole'', ''Adrian'' INTO contact.lname, contact.fname;
```

Можно сразу присвоить значение всей переменной типа ROWTYPE (всем ее полям), если в правильном порядке указать значения для всех столбцов:

```
DECLARE
    product item%ROWTYPE;

BEGIN
    select NULL, ''Widget'', ''1.45'', ''1.99'' into product;
END;
```

Если присваиваемые значения и переменные имеют разные типы данных, то во всех возможных случаях PostgreSQL осуществит приведе-

ния типов. В последнем примере значения себестоимости и отпускной цены принадлежат типу NUMERIC(7,2), но им успешно присвоены значения, введенные как текстовые.

Присваивания выполняются сервером PostgreSQL как операторы SELECT, даже если используется форма :=. Можно применить SELECT для присваивания переменным значений из базы данных. Для этого включим в оператор выражение FROM и (необязательно) условие WHERE. Например:

```
SELECT * INTO product FROM item WHERE item_id = 9;
```

Необходимо позаботиться о том, чтобы SELECT возвращал только одну строку, т. к. дополнительные строки будут молча отброшены, лишь строка, возвращенная первой, будет присвоена перечисленным переменным. Чуть позже будет рассказано о том, как написать программу так, чтобы она выполнялась для каждой строки оператора SELECT, возвращающего несколько строк.

Может случиться так, что SELECT не возвратит ни одной строки, тогда присваивание не состоится. В PostgreSQL существует специальная булева переменная FOUND, которая становится доступной непосредственно после выполнения присваивания с помощью оператора SELECT INTO. По ее значению можно определить, успешно ли прошло присваивание:

```
SELECT * INTO product FROM item WHERE description ~~ ''%Cube%'';
IF NOT FOUND THEN
-- take some recovery action
END IF;
```

PERFORM

Иногда не требуется получать данные, представляющие собой результат выполнения оператора SELECT, например, если он применяется для вызова функции, имеющей побочные эффекты. В подобной ситуации можно оценить выражение или запрос и отбросить данные с помощью PERFORM:

```
PERFORM <запрос>;
```

По существу оператор PERFORM выполняет запрос SELECT посредством диспетчеров SPI и игнорирует результат.

Управляющие структуры

PL/pgSQL предоставляет структуры, управляющие последовательностью действий внутри функции: операторы ветвления, возврата, условный оператор и оператор цикла.

Возвращение из функций

Возврат значения из функции осуществляется при помощи оператора RETURN:

```
RETURN <выражение>;
```

Выполнение функции останавливается после получения значения этого выражения. Значение выражения становится доступным оператору, вызвавшему функцию, в качестве результата работы функции. Значение должно быть совместимо с типом возврата, объявленного для функции, а в случае необходимости осуществляется приведение типа.

В PL/pgSQL функция должна возвращать значение, и если конец самого внешнего блока функции будет достигнут, а RETURN при этом не будет выполнен, то возникнет ошибка времени выполнения.

Исключения и сообщения

Выполнение функции также может быть остановлено, если наступит какая-то ситуация, делающая продолжение работы невозможным. Функция может не только возвращать значение, но и порождать исключение. Исключение служит причиной для внесения записи в журнал PostgreSQL и может привести к немедленному завершению хранимой процедуры:

```
RAISE level 'format' [, variable ...];
```

Оператор RAISE регистрирует в журнале исключение, которое может характеризоваться разной степенью серьезности.

PostgreSQL определяет три уровня (табл. 10.11):

Таблица 10.11. Уровни исключений

Уровень	Поведение
DEBUG (отладочное)	Записывает сообщение в журнал (обычно отключено)
NOTICE (примечание)	Записывает сообщение в журнал и посылает его в приложение
EXCEPTION (исключение)	Записывает сообщение в журнал и завершает хранимую процедуру

Уровень DEBUG полезен для сбора дополнительной информации во время разработки. Уровень NOTICE выдает предупреждения о наличии исправимых ошибок. В случае необходимости предупреждения можно сделать доступными клиентским приложениям при помощи механизма NOTIFY. Уровень EXCEPTION предназначен для неисправимых ошибок, когда хранимая процедура не может продолжить свою работу. За подробной информацией обращайтесь к руководству программиста по PostgreSQL.

Строка format определяет формат сообщения об ошибке. Внутри этой строки каждый символ % последовательно заменяется значениями переменных. В отличие от printf в C, в операторе RAISE можно использовать только идентификаторы, но не выражения.

Рассмотрим оператор:

```
RAISE DEBUG '' The value of n is %'', n;
```

В системный журнал PostgreSQL, /usr/local/pgsql/data/postmaster.log, вносится запись, выглядящая примерно так:

```
DEBUG: The value of n is 4
```

Рассмотрим в качестве иллюстрации процедуру:

```
create function scope() returns integer AS '
BEGIN
DECLARE
n integer := 4;
BEGIN
RAISE DEBUG ''n is %'', n;
return n;
END;
' language 'plpgsql';
```

Если выполнить эту хранимую процедуру в psql, сообщение не будет выведено:

```
bpfinal=# SELECT scope();
 scope
-------
     4
(1 row)

bpfinal=#
```

А вот если увеличить уровень серьезности оператора RAISE до NOTICE, то в psql, как и в системном журнале, появится сообщение:

```
bpfinal=# SELECT scope();
NOTICE:  n is 4
 scope
-------
     4
(1 row)

bpfinal=#
```

И наконец, на уровне EXCEPTION хранимая процедура преждевременно завершается с ошибкой:

```
bpfinal=# SELECT scope();
ERROR:  n is 4

bpfinal=#
```

Условные операторы

PL/pgSQL поддерживает несколько видов условных операторов, то есть логических структур, выполняющих один из двух или более наборов операторов или возвращающих один из двух или более возможных результатов на основе некоторой проверки. Может быть, они являются самой полезной частью PL/pgSQL. Самым распространенным условным оператором, как и во многих других языках программирования, является, наверное, оператор IF.

IF–THEN–ELSE

```
IF expression
THEN
statements
[ELSE
      statements]
END IF;
```

Если выражение expression оценено как TRUE, то выполняются операторы из части THEN оператора IF. В противном случае, если присутствует необязательная часть ELSE, то выполняются входящие в нее операторы.

Как и в других языках программирования, разрешено вложение операторов IF, то есть включение дополнительных операторов IF в часть THEN или ELSE.

NULLIF и CASE

Существуют также две условные функции SQL, возвращающие значение в зависимости от результатов некоторой проверки. Это функции NULLIF и CASE, в PostgreSQL они могут применяться в хранимых процедурах и в обычном SQL.

Функция NULLIF возвращает NULL, если значение input соответствует заданному значению value, а в случае несоответствия возвращает input неизмененным:

```
NULLIF(input, value)
```

Функция вернет NULL, если значение выражения input = value равно TRUE, в противном случае будет возвращена величина input.

Функция CASE выбирает одно из нескольких значений в зависимости от введенной величины. Синтаксис функции выглядит так:

```
CASE
      WHEN expression
      THEN expression
      ...
ELSE expression
END;
```

Пар WHEN/THEN может быть столько, сколько нужно. Выражение в целом возвращает результат выполнения части THEN, соответствующей первому WHEN, результат выполнения которого оказался равен TRUE. Если нет соответствия ни одной из частей WHEN, то вычисляется выражение ELSE. Например, в результате выполнения оператора, приведенного ниже, значение res будет равно 5, 6 или 7 в зависимости от того, какое значение имеет n2 – 1, 2 или какое-то другое:

```
res := CASE
        WHEN n2 = 1
        THEN 5
        WHEN n2 = 2
        THEN 6
        ELSE 7
END;
```

Циклы

PL/pgSQL обладает широким набором возможностей по организации циклов (циклы также называют итерационными управляющими структурами).

Простейшим циклом является неуправляемый цикл, выполняемый бесконечно, если не прервать его оператором EXIT:

```
[<<label>>]
LOOP
    statements
END LOOP;
```

Все структуры LOOP в PL/pgSQL могут быть помечены, как и блоки BEGIN ... END. Такая метка выступает в качестве цели для оператора EXIT, вызывающего завершение указанного цикла. Для читателей, знакомых с языком C, приведем аналог – многоуровневую версию break – то, чего не хватает в C:

```
The EXIT statement

EXIT [label] [WHEN expression];
```

Цикл LOOP, помеченный меткой label, завершается, а выполнение программы продолжается с оператора, следующего непосредственно после конца цикла. Метка должна относиться к текущему циклу LOOP или к охватывающему LOOP. Если метка не указана, то завершается текущий цикл. Если задано выражение WHEN, то EXIT не выполняется до тех пор, пока значение expression не становится равным TRUE.

Вот пример бесконечного цикла :

```
<<infinite>>
LOOP
```

```
        n := n + 1;
        EXIT infinite WHEN n >= 10;
END LOOP;
```

Более управляемым является цикл WHILE, исполняющий ряд операторов до тех пор, пока условие остается истинным.

Цикл WHILE

```
[<<label>>]
WHILE expression
LOOP
        statements
END LOOP;
```

Можно добиться того, чтобы цикл выполнялся фиксированное количество раз, используя оператор цикла FOR:

```
FOR name IN [REVERSE] from .. to
LOOP
        statements
END LOOP;
```

В этом типе цикла его тело выполняется один раз для каждого значения из диапазона, определенного целыми выражениями from и to. Для цикла создается новая переменная с именем name. Ей последовательно присваиваются все значения из диапазона, при каждом выполнении тела цикла значение увеличивается на единицу. REVERSE задает обратный порядок работы цикла, переменная name уменьшается на единицу при каждом выполнении тела цикла.

Цикл FOR

Приведем простой пример работы цикла FOR:

```
FOR cid IN 1 .. 15
LOOP
    SELECT * INTO row FROM customer
WHERE customer_id = cid;
    -- обработка клиента
END LOOP;
```

Цикл выполняется 15 раз, при этом переменная cid принимает значения от 1 до 15, а клиенты по одному выбираются в переменную row. Нижняя и верхняя границы значений переменной цикла могут быть выражениями, поэтому можно просмотреть всю таблицу customer, а не только первых 15 клиентов при помощи:

```
SELECT COUNT(*) INTO ncustomers FROM customer;
FOR cid IN 1 .. ncustomers
...
```

Однако более аккуратным решением было бы применение альтернативной формы цикла FOR, позволяющей выполнять цикл один раз для каждой строки таблицы, а в действительности даже для каждой строки, возвращенной произвольным оператором SELECT:

```
FOR row IN SELECT ...
LOOP
      statements
END LOOP;
```

Для каждой строки, возвращенной SELECT, переменной row присваивается значение и выполняются операторы statements. Переменная row, отведенная для хранения строки, должна быть предварительно объявлена как принадлежащая к типу ROWTYPE или RECORD. Последняя обработанная строка останется доступной и после того, как цикл закончится или будет завершен при помощи EXIT.

Представленная ниже процедура, будучи запущенной в psql, выводит фамилии всех клиентов:

```
DECLARE
    row record;
BEGIN
    FOR row IN SELECT * FROM customer
    LOOP
    RAISE NOTICE ''Family Name is %'', row.lname;
    END LOOP;
END:
```

Итак, все программные структуры PL/pgSQL рассмотрены, и пришло время приступить к их применению.

Попробуйте сами. Хранимая процедура

Предположим, мы задумались о том, как использовать нашу учебную базу данных для того, чтобы содействовать заказу дополнительных изделий у поставщиков по мере опустошения склада. В базе уже есть таблица stock, которая отслеживает количество изделий, доступных для продажи. Хотелось бы иметь возможность применить эту информацию для автоматического формирования заказов поставщикам.

Ниже приведена процедура, реализующая первый шаг на пути достижения поставленной цели – автоматической подачи повторных заказов. Функция reorders ищет в таблице stock изделия, количество которых на складе меньше некой определенной величины. Для каждого такого изделия она вносит запись в новую таблицу reorders.

Следующим шагом (он оставлен читателям в качестве упражнения) должно стать генерирование заказов на основе таблицы reorders:

```
-- Удалить и создать временную таблицу для формирования заказов
drop table reorders;
create table reorders
(
  item_id    integer,
  message   text
);

-- reorders
-- просмотреть таблицу stock, чтобы повторно заказать изделия, которых
осталось мало
create function reorders(int4) returns integer as '
declare
    min_stock alias for $1;
    reorder_item integer;
    reorder_count integer;
    stock_row stock%rowtype;
    msg text;
begin
    select count(*) into reorder_count from stock
           where quantity <= min_stock;
    for stock_row in select * from stock
            where quantity <= min_stock
    loop
        declare
            item_row item%rowtype;
        begin
            select * into item_row from item
                    where item_id = stock_row.item_id;
            msg = ''order more '' ||
                         item_row.description || ''s at '' ||
                         to_char(item_row.cost_price,''99.99'');
            insert into reorders
                         values (stock_row.item_id, msg);
        end;
    end loop;
    return reorder_count;
end;
' language 'plpgsql';
```

Сохраните приведенный выше код в `sproc.sql` (также можно загрузить его с сайта Wrox, расположенного по адресу *http://www.wrox.com*).

Создав функцию и выполнив ее (при этом минимальное количество изделий на складе определено как 3), мы получаем результат 3, означающий, что уровень запасов трех изделий не превышает 3 единиц:

```
bpfinal=# \i sproc.sql
DROP
CREATE
```

```
CREATE
bpfinal=# SELECT reorders(3);
 reorders
------------
          3
(1 row)

bpfinal=#
```

В таблицу reorders были внесены идентификаторы тех трех изделий, запас которых иссякает:

```
bpfinal=# SELECT * SELECT reorders;
 item_id |               message
---------+-------------------------------------
       2 | order more Rubik Cubes at   7.45
       5 | order more Picture Frames at   7.54
      10 | order more Carrier Bags at    .01
(3 rows)

bpfinal=#
```

Можно было применить для создания таблицы reorders команду CREATE TEMPORARY TABLE, в этом случае таблица была бы автоматически удалена по завершении сессии. Мы же выбрали команду CREATE TABLE, поэтому таблица reorders будет сохранена. Дело в том, что она может еще понадобиться, если приложение, реализующее повторный заказ, отделено от проверки уровня запасов.

Можно проверить результат, самостоятельно обратившись к таблице stock:

```
bpfinal=# SELECT * FROM stock;
 item_id | quantity
---------+-----------
       1 |       12
       2 |        2
       4 |        8
       5 |        3
       7 |        8
       8 |       18
      10 |        1
(7 rows)
bpfinal=#
```

Как это работает

Функция reorders использует оператор SELECT для выбора всех строк таблицы stock, имеющих низкий уровень запасов изделия. Цикл применяется для перебора строк, возвращаемых оператором SELECT. Тело цикла, в котором INSERT служит для внесения информации в таблицу reorders, выполняется для каждой строки итогового множества. Возврат резуль-

тата из reorders обеспечивается другим оператором SELECT, который считает количество соответствующих строк таблицы stock.

Чтобы завершить систему автоматической повторной подачи заказов, надо было бы выделить изделия и разместить заказы у поставщиков. Вероятно, следовало бы установить минимальный уровень запасов для каждого изделия и внести его в таблицу item. Аналогично, количество изделий, которое требуется заказать, можно сделать переменным, зависящим, например, от истории продаж или каких-то сезонных факторов.

Динамические запросы

Обычно запросы к базе данных в хранимых процедурах являются фиксированными или просто параметризованными. К таблице в большинстве случаев обращаются за строками, некий столбец которых соответствует заданному значению, или же чтобы обновить строку, задав новые значения для столбцов. Это может быть достигнуто с помощью операторов SELECT и UPDATE и подстановкой значений переменных процедуры:

```
INSERT INTO reorders VALUES (stock_row.item_id, msg);
```

В некоторых редких случаях может потребоваться использовать значение переменной для определения имени таблицы или столбца в некоторой операции. PostgreSQL не предоставляет такой возможности, поскольку СУБД необходимо оптимизировать запрос только один раз, а не при каждом его выполнении.

А вот PL/pgSQL поддерживает оператор EXECUTE, который позволяет выполнить произвольный оператор SQL, указанный в виде символьной строки:

```
EXECUTE query-string
```

Строка для запроса может динамически создаваться внутри хранимой процедуры с помощью строковых операторов, представленных ранее.

Особое внимание стоит уделить правильному употреблению кавычек для имен и буквенных значений внутри символьной строки. Обеспечить корректность могут две функции: названия всех таблиц и столбцов следует обработать функцией quote_ident, которая генерирует строку, подходящую для формирования части запроса, а значения должны быть обработаны функцией quote_value.

Приведем пример, в котором создается универсальное обновление, использующее переменные для имен и значений:

```
EXECUTE ''UPDATE ''
|| quote_ident(tablename)
|| '' SET ''
    || quote_ident(columnname)
```

```
    || '' = ''
    || quote_literal(columnvalue)
    || '' WHERE ''
...;
```

Знайте, что такой способ доступа к базе данных может оказаться неэффективным, т. к. создаваемые запросы должны интерпретироваться и планироваться при каждом исполнении.

Динамические запросы также могут применяться в циклах FOR вместо оператора SELECT для произведения итераций над записями. Синтаксис данного варианта таков:

```
FOR row IN EXECUTE query-string
LOOP
      statements
END LOOP;
```

Функции SQL

Темой данной главы является применение PL/pgSQL для создания хранимых процедур, но необходимо отметить, что можно создавать функции и с помощью SQL. Для этого надо указать в качестве языка процедуры 'sql' и использовать SQL-операторы PostgreSQL вместо операторов PL/pgSQL.

Как и в PL/pgSQL, SQL-функции принимают параметры, на которые можно ссылаться как на $1, $2 и т. д. Определенная в функции переменная $1 при вызове функции автоматически заменяется первым аргументом и т. д. Управляющих структур не существует, применяются только SQL-операторы PostgreSQL. Другими словами, в PL/pgSQL существуют такие возможности, как переменные, условные вычисления и организация циклов, а в SQL функции поддерживают только замену аргументов. Значение, возвращаемое SQL-функцией, – это данные, полученные как результат последнего выполненного оператора SQL, обычно оператора SELECT.

Преимущество использования SQL-функций для хранимых процедур заключается в том, что нет необходимости загружать в базу данных обработчик языка PL/pgSQL.

Особенность SQL-функций в том, что они позволяют вернуть более чем одну строку данных. Если объявить тип возврата функции как setof, а затем применить соответствующий SELECT, можно вернуть несколько строк. Вот функция, которая возвращает всех клиентов, живущих в определенном городе:

```
CREATE function sqlf(text) RETURNS setof customer AS '
    SELECT * FROM customer WHERE town = $1;
' language 'sql';
```

Запустив функцию в psql, мы увидим, что будут возвращены три строки:

```
bpfinal=# SELECT sqlf('Bingham');
 ?column?
-------------
 136842856
 136842856
 136842856
(3 rows)

bpfinal=#
```

К сожалению, в psql нет возможности обрабатывать сразу целую строку, поэтому необходимо выбирать столбцы по одному. Чтобы сделать это, применим для извлечения нужного столбца такой синтаксис: column(function()). Здесь перечисляются фамилии клиентов, живущих в городе Bingham, и дается имя столбцу результатов:

```
bpfinal=# SELECT lname(sqlf('Bingham')) AS customer;
 customer
------------
 Stones
 Stones
 Jones
(3 rows)

bpfinal=#
```

Триггеры

В примере хранимой процедуры была создана функция, позволявшая определять, для каких изделий требуется пополнение запасов. Функция записывала сообщения-напоминания в таблицу reorders. Для того чтобы она приносила пользу, необходимо обеспечить регулярное выполнение процедуры, например раз в сутки, во время ночной пакетной обработки. Было бы очень удобно, если бы удалось найти способ автоматического обновления таблицы reorders, вместо того чтобы программировать клиентские приложения так, чтобы обновлять записи.

В главе 8 был представлен принцип ссылочной целостности, гарантирующий, что данные в базе всегда имеют смысл. Например, если удаляется клиент, необходимо обеспечить одновременное удаление относящейся к нему истории заказов. Было показано, что достичь такой целостности в PostgreSQL можно при помощи ограничений.

Но в некоторых приложениях недостаточно применения ограничений: предположим, что необходимо предотвратить удаление клиента, у которого еще есть невыполненные заказы, если же все заказы уже отправлены, то клиента можно удалять.

В главе 8 было показано, как посредством ограничений для таблиц и столбцов можно вводить более сложные правила для целостности данных, но, по существу, все эти правила были статическими. Можно было указать, что должна существовать связанная строка, или запретить удаление в случае существования связанных строк, но не было никакого способа задать сложные условия (например, строка не может существовать до тех пор, пока некоторое другое условие не станет истинным). Нельзя было и осуществлять более сложные операции, определяемые пользователем, при добавлении или удалении строк.

Решением проблемы являются триггеры. С помощью триггера можно сделать так, чтобы PostgreSQL выполняла хранимую процедуру в случае наступления какого-либо события, такого как применения к таблице оператора INSERT, DELETE или UPDATE.

Комбинация хранимых процедур и триггеров дает возможность реализовывать сложную бизнес-логику непосредственно в базе данных. Как говорилось ранее, лучшим местом для определения бизнес-логики является именно база данных.

Для применения триггера сначала надо определить триггерную процедуру, а затем уже создавать сам триггер, определяющий, когда эта процедура должна выполняться.

Создание триггеров

Триггеры создаются при помощи команды CREATE TRIGGER, которая имеет такой синтаксис:

```
CREATE TRIGGER name { BEFORE | AFTER }
    { event [OR ...] }
    ON table FOR EACH { ROW | STATEMENT }
    EXECUTE PROCEDURE func ( arguments )
```

В данном случае event – это оператор INSERT, DELETE или UPDATE.

Триггер говорит: «Запускать данную хранимую процедуру каждый раз, когда такое-то событие происходит с такой-то таблицей».

Триггеру дается название, используемое при удалении триггера, который больше не нужен:

```
DROP TRIGGER name ON table;
```

Триггер срабатывает, когда происходит определенное событие – DELETE, INSERT или UPDATE. Можно сделать так, чтобы триггер запускался после того, как событие произошло, тогда хранимая процедура получит доступ и к исходным данным (для обновлений и удалений), и к новым (для вставок и обновлений). А можно запускать триггер до того, как событие произойдет. Тогда можно предотвратить обновление или изменить вставляемые или обновляемые данные.

Можно указать несколько событий, разделяя их условием `OR`.

Как известно, некоторые операторы SQL могут воздействовать на несколько строк данных. Если запуск триггера вызван многострочным обновлением, можно выбрать способ запуска триггера: для каждой обновляемой строки или один раз для всего обновления. Указываем `ROW`, если нужно, чтобы триггер запускался несколько раз, и `STATEMENT` – в противном случае.

Аргументы, передаваемые в функцию, могут служить для распознавания похожих триггеров, поэтому одна функция может быть использована для нескольких триггеров.

Для автоматизации обновления таблицы повторных заказов `reorders` можно создать хранимую процедуру под названием `reorder_trigger` и триггер, который вызывал бы ее при каждом изменении таблицы `stock`:

```
CREATE TRIGGER trig_reorder
AFTER INSERT OR UPDATE ON stock
FOR EACH ROW EXECUTE PROCEDURE reorder_trigger(3);
```

Имейте в виду, что триггерная процедура (`trig_reorder` в данном случае) должна быть определена до того, как будет создан сам триггер.

Аргумент триггерной процедуры применяется для передачи минимального уровня запасов изделия, который в нашем случае равен 3.

Триггерные процедуры

Триггер срабатывает, когда выполняется некоторое условие, и выполняет специальную хранимую процедуру, называемую триггерной. Триггерная процедура очень похожа на хранимую, но она чуть более ограниченна, что объясняется способом обращения к ней.

Триггерная процедура создается как функция без параметров, имеющая специальный тип возврата – `OPAQUE`. Тип возврата `OPAQUE` применяется для функций, возвращающих такие значения, которые PostgreSQL не может обработать непосредственно.

PostgreSQL вызовет триггерную процедуру, когда над некоторой конкретной таблицей будут производиться изменения. Процедура должна вернуть или `NULL`, или строку, соответствующую структуре таблицы, для которой была вызвана триггерная процедура.

Для триггеров `AFTER`, вызываемых после `UPDATE`, обычно рекомендуется, чтобы триггерная процедура возвращала `NULL`.

В случае триггеров `BEFORE` возвращенный результат используется для управления производимым обновлением. Если триггерная процедура возвращает `NULL`, то `UPDATE` не выполняется, если же возвращается строка данных, то она выступает как источник обновления, давая триггерной процедуре возможность изменить данные до того, как они будут зафиксированы в базе.

Попробуйте сами. Триггеры

Рассмотрим в качестве примера простую триггерную процедуру, обновляющую таблицу reorders после изменения запасов:

```
Create function reorder_trigger() returns opaque AS '
declare
    mq integer;
    item_record record;
begin
    mq := tg_argv[0];
    raise notice ''in trigger, mq is %'', mq;
    if new.quantity <= mq
        then
          select * into item_record from item
          where item_id = new.item_id;
        insert into reorders
          values (new.item_id, item_record.description);
    end if;
    return NULL;
end;
' language 'plpgsql';
```

Итак, у нас есть процедура и триггер, который ее использует, и теперь можно посмотреть на него в действии (посредством psql).

Загрузим определение функции и триггера из файла сценария:

```
bpfinal=# \i sproc.sql
...
CREATE
CREATE

bpfinal=#
```

Затем попробуем изменить уровень запасов изделия так, чтобы оно уменьшилось до 3 или еще больше:

```
bpfinal=# UPDATE stock SET quantity = 3 WHERE item_id = 1;
NOTICE:  in trigger, mq is 3
UPDATE 1

bpfinal=#
```

Как видите, был запущен триггер, который выдал уведомление. Обновление таблицы stock продолжается, но триггер обновляет и таблицу reorders, добавляя новую строку:

```
bpfinal=# SELECT * FROM reorders;
 item_id |   message
---------+-------------
       1 | Wood Puzzle
```

```
(1 row)

bpfinal=#
```

Как это работает

Триггерная процедура вызывается каждый раз, когда к таблице stock применяется оператор INSERT или UPDATE. Она проверяет уровень запасов изделия после внесения изменений и, если он ниже минимального количества, добавляет запись в таблицу reorders.

Вероятно, вы обратили внимание на некоторые особенности применения триггерных процедур. Во-первых, в триггерных процедурах к аргументам не обращаются как к $1, $2 и т. д. Доступ к аргументам обеспечивается посредством специальных переменных, к которым триггерные процедуры имеют доступ автоматически. Аргументы передаются в массиве с именем tg_argv начиная с tg_argv[0].

Во-вторых, внутри триггерных процедур существуют две специфических записи, OLD и NEW. Для триггеров ROW они содержат данные из строки, измененяемой оператором, запустившим триггер. Несложно догадаться, что OLD содержит данные до обновления, а NEW – новые данные после изменения (или предложенного изменения в триггере BEFORE).

В триггерных процедурах в наличии имеются следующие специальные переменные (табл. 10.12):

Таблица 10.12. Специальные переменные триггерных процедур

Переменная	Описание
NEW	Запись, содержащая новую строку базы данных
OLD	Запись, содержащая старую строку базы данных
TG_NAME	Текстовая переменная, содержащая название сработавшего триггера, который вызвал запуск триггерной процедуры
TG_WHEN	Текстовая переменная, содержащая текст 'BEFORE' или 'AFTER', в зависимости от типа триггера
TG_LEVEL	Текстовая переменная, содержащая 'ROW' или 'STATEMENT', в зависимости от определения триггера
TG_OP	Текстовая переменная, содержащая 'INSERT', 'DELETE' или 'UPDATE', в зависимости от того, какое событие вызвало запуск данного триггера
TG_RELID	Идентификатор объекта, представляющий таблицу, для которой был активирован триггер
TG_RELNAME	Название таблицы, для которой был активирован триггер
TG_NARGS	Целая переменная, содержащая количество аргументов, указанных в определении триггера
TG_ARGV	Массив символьных строк, содержащий параметры процедуры, начиная с нуля. По недействительным индексам массива возвращаются NULL-значения

В заключение рассмотрим другую триггерную функцию, запрещающую удаление клиента, не все заказы которого выполнены. Проверяем, является ли значение столбца `date_shipped` таблицы `orderinfo` равным `NULL` для каких-либо заказов, размещенных клиентом, которого пытаются удалить, и запрещаем удаление. Если незаконченных заказов нет, можно продолжить процесс удаления, но при этом необходимо «вычистить» информацию о заказах, размещенных клиентом в прошлом, а также данные об изделиях, входящих в такие заказы.

Попробуйте сами. Еще один триггер

```
create function customer_trigger() returns opaque AS
declare order_record record;
begin
    -- удаление клиента
    -- запретить, если есть невыполненные заказы
    select * into order_record from orderinfo
           where customer_id = old.customer_id
           and date_shipped = NULL;
    if not found
    then
        -- OK, можно удалять клиента
        raise notice ``удаление разрешено - нет невыполненных заказов'';

        -- для ссылочной целостности следует привести все в порядок
        -- необходимо удалить все выполненные заказы
        -- но сначала удалим информацию о заказах
        for order_record in select * from orderinfo
            where customer_id = old.customer_id
        loop
            delete from orderline
                   where orderinfo_id = order_record.orderinfo_id;
        end loop;

        -- теперь удалим записи о заказах
        delete from orderinfo
               where customer_id = old.customer_id;

        -- возврат старой записи для разрешения удаления клиента
        return old;
    else
        -- есть заказы, для предотвращения удаления возвратить NULL
        raise notice ``удаление запрещено - имеются невыполненные заказы'';
        return NULL;
    end if;
end;
' language 'plpgsql';

create trigger trig_customer before delete on customer
for each row execute procedure customer_trigger();
```

Для того чтобы посмотреть на поведение триггера, выпо---лним его.

Сначала сделаем так, чтобы один из заказов имел значение NULL для даты отправки. Так мы покажем, что он еще не завершен:

```
bpfinal=# UPDATE orderinfo SET date_shipped = NULL WHERE orderinfo_id = 3;
UPDATE 1
bpfinal=# SELECT * FROM orderinfo;
 orderinfo_id | customer_id | date_placed | date_shipped | shipping
--------------+-------------+-------------+--------------+----------
            1 |           3 | 2000-03-13  | 2000-03-17   |     2.99
            2 |           8 | 2000-06-23  | 2000-06-24   |     0.00

            4 |          13 | 2000-09-03  | 2000-09-10   |     2.99
            5 |           8 | 2000-07-21  | 2000-07-24   |     0.00
            3 |          15 | 2000-09-02  |              |     3.99
(5 rows)

bpfinal=#
```

Теперь, если попытаться удалить клиента с номером 15, чей заказ номер 3 ожидает исполнения, то будет выдано уведомление о наличии незавершенных заказов, и строки не будут удалены, поскольку продолжать выполнение операций не разрешено:

```
bpfinal=# DELETE FROM customer WHERE customer_id = 15;
NOTICE: удаление запрещено - имеются невыполненные заказы
DELETE 0

bpfinal=#
```

Удаление клиентов, у которых нет невыполненных заказов, не представляет никаких трудностей после того, как предварительно удалены записи в таблице orderinfo. Дело в том, что для этой таблицы задано ограничение, запрещающее удаления именно такого вида. Триггер заботится об этом, тем самым реализуя другой подход к обеспечению ссылочной целостности.

У клиента номер 3 все заказы выполнены, поэтому можно удалять его, предоставив триггеру сделать всю работу:

```
bpfinal=# DELETE FROM customer WHERE customer_id = 3;
NOTICE: удаление разрешено - нет невыполненных заказов
DELETE 1

bpfinal=#
```

Проверка таблиц показывает, что никаких следов удаленного клиента обнаружить не удается:

```
bpfinal=# SELECT * FROM orderinfo;
 orderinfo_id | customer_id | date_placed | date_shipped | shipping
--------------+-------------+-------------+--------------+----------
            2 |           8 | 2000-06-23  | 2000-06-24   |     0.00
            4 |          13 | 2000-09-03  | 2000-09-10   |     2.99
```

```
           5 |           8 | 2000-07-21  | 2000-07-24   |     0.00
           3 |          15 | 2000-09-02  |              |     3.99
(4 rows)

bpfinal=# SELECT * FROM orderline;
 orderinfo_id | item_id | quantity
--------------+---------+----------
            2 |       1 |        1
            2 |      10 |        1
            2 |       7 |        2
            2 |       4 |        2
            3 |       2 |        1
            3 |       1 |        1
            4 |       5 |        2
            5 |       1 |        1
            5 |       3 |        1
(9 rows)

bpfinal=#

bpfinal=# SELECT customer_id, fname, lname FROM customer;
 customer_id |   fname   |  lname
-------------+-----------+-----------
           1 | Jenny     | Stones
           2 | Andrew    | Stones
           4 | Adrian    | Matthew
           5 | Simon     | Cozens
           6 | Neil      | Matthew
           7 | Richard   | Stones
           8 | Ann       | Stones
           9 | Christine | Hickman
          10 | Mike      | Howard
          11 | Dave      | Jones
          12 | Richard   | Neill
          13 | Laura     | Hendy
          14 | Bill      | O'Neill
          15 | David     | Hudson
(14 rows)
bpfinal=#
```

Зачем нужны хранимые процедуры и триггеры?

Причин использовать хранимые процедуры и триггеры почти так же много, как приложений, работающих с базами данных. Приведем некоторые из них:

- **Централизованная проверка достоверности**

  Все условия для обновления таблиц можно ввести в действие в одном месте, не зависящем от клиентских приложений. Изменить условия можно только в одном месте.

- **Отслеживание изменений**

  Посредством триггера можно создать журнал аудита, записи которого заносятся в другую таблицу при обновлении строк, при этом записываться может имя пользователя, внесшего изменение, время и дата изменения, можно также регистрировать изменившиеся данные.

- **Обеспечение большей безопасности**

  Используя переменную PostgreSQL `current_user`, вы повысите безопасность.

- **Отсроченные удаления**

  С помощью триггера можно пометить, что строки должны быть удалены не в тот момент, когда приложение пытается это сделать, а позже.

- **Преобразование данных для пользователей**

  Применение триггеров и хранимых процедур позволяет создать более простую, однотабличную версию неких данных, которую пользователям будет проще редактировать. Например, можно создать таблицу и связать ее с Microsoft Excel. При изменении строк в этой таблице обновляются строки в «настоящих» таблицах, которые (вероятно) имеют более сложную структуру.

Резюме

В данной главе были рассмотрены способы расширения функциональности запросов PostgreSQL. Было показано, что PostgreSQL предоставляет множество операторов и функций, которые можно использовать для уточнения запросов и извлечения информации.

Процедурные языки, поддерживаемые PostgreSQL, позволяют совершенствовать обработку со стороны сервера за счет написания процедур на PL/pgSQL, SQL и других языках. Для сервера базы данных появляется благоприятная возможность реализовать в приложениях сложную функциональность вне зависимости от клиента.

Хранимые процедуры хранятся в самой базе данных. Они могут вызываться приложением или же, в форме триггеров, вызываться автоматически при внесении изменений в таблицы базы данных. Таким образом, появляются новые средства обеспечения ссылочной целостности.

В случае простой ссылочной целостности обычно лучше применять ограничения, т. к. они проще, рациональнее и меньше подвержены ошибкам. Мощь триггеров и хранимых процедур вступает в дело тогда, когда декларативные ограничения становятся достаточно трудными для понимания или когда требуется написать ограничение, слишком сложное для декларативной формы.

11

Администрирование PostgreSQL

В этой главе рассказывается о том, как обслуживать базу данных PostgreSQL. Начнем с чуть более подробного, чем в главах 3 и 4, рассмотрения установки PostgreSQL, производимой по умолчанию. Затем перейдем к обслуживанию, необходимому для того, чтобы получить от базы данных максимум возможного.

Советуем создать для проведения экспериментов тестовую базу данных. Также установите приложение `pgAdmin`, предоставляющее графический пользовательский интерфейс для функций администрирования из клиента Microsoft Windows. В главе 5 подробно рассказано о `pgAdmin`.

Для эффективного развертывания и сопровождения реляционной базы данных необходимо выполнить ряд действий, которые и будут обсуждаться в данной главе:

- Управление сервером
  - Запуск, остановка и управление серверным процессом базы данных
  - Удаленный доступ (по сети)
  - Работа с системными журналами
- Доступ пользователей
  - Создание и удаление учетных записей пользователей
  - Контроль прав доступа
- Сопровождение данных
  - Создание и удаление баз данных и схем

    - Резервное копирование и восстановление данных
    - Установка новых версий PostgreSQL
- Безопасность
- Параметры конфигурации
- Производительность и настройка
    - Создание индексов
    - Мониторинг и настройка

Установка по умолчанию

Если скомпилировать PostgreSQL из исходного кода и установить, не изменив ни одного из параметров, заданных по умолчанию, то в каталог `/usr/local/pgsql` будет инсталлирована уже знакомая нам система.

Если устанавливается версия из пакета, то может оказаться, что она инсталлируется в другое место, например `/var/lib/pgsql`, но структура каталогов внутри каталога установки будет в целом такой же, только исполняемые программы могут попасть в другой каталог, такой как `/usr/bin`.

В каталог установки входят семь подкаталогов, предназначенных для хранения программ, библиотек, заголовочных файлов PostgreSQL и, конечно же, самой базы данных.

Перечислим эти каталоги:

- `bin`
- `include`
- `lib`
- `doc`
- `man`
- `share`
- `data`

bin

Каталог `bin` содержит программы (табл. 11.1), из которых собственно и состоит PostgreSQL:

Таблица 11.1. Программы, входящие в каталог bin

Программа	Описание
`postgres`	Сервер базы данных
`postmaster`	Процесс, принимающий соединения (тот же исполняемый файл, что и `postgres`)

Программа	Описание
psql	Утилита командной строки для PostgreSQL
initdb	Утилита инициализации СУБД
pg_ctl	Управление PostgreSQL – запуск, остановка и перезапуск сервера
createuser	Утилита создания пользователя базы данных
dropuser	Утилита удаления пользователя базы данных
createdb	Утилита создания базы данных
dropdb	Утилита удаления базы данных
initlocation	Утилита создания дополнительных областей для базы данных
pg_dump	Утилита резервного копирования базы данных
pg_dumpall	Утилита резервного копирования всех баз данных
pg_restore	Утилита восстановления базы данных из резервной копии
pg_upgrade	Утилита, помогающая обновлять версии PostgreSQL
pg_passwd	Утилита для работы с файлами паролей, специфичными для сервера
vacuumdb	Утилита, помогающая оптимизировать базу данных (см. раздел «Производительность» далее)
ipcclean	Утилита, удаляющая сегменты совместно используемой памяти после аварийного завершения
pg_config	Утилита для определения конфигурации PostgreSQL
createlang	Утилита добавления поддержки языков (см. главу 10)
droplang	Утилита удаления языковой поддержки
ecpg	Встроенный компилятор SQL (см. главу 14)
pg_id	Утилита, идентифицирующая пользователя операционной системы

include и lib

Каталоги include и lib содержат все заголовочные файлы и библиотеки, необходимые для создания клиентских приложений PostgreSQL. Здесь можно найти библиотеки, необходимые для сборки libpq приложений и программ на C, содержащих встроенный SQL. Более подробно о libpq и ecpg рассказано в главах 13 и 14.

doc

В каталоге doc находится документация по PostgreSQL. Вероятно, самым полезным документом является doc/html/index.html – начальная страница документации, которую можно прочитать в броузере.

man

Каталог man включает страницы учебника, подобного руководству по UNIX. Добавив этот каталог в переменную окружения MANPATH, вы сможете использовать стандартную команду man для получения доступа к данным страницам:

```
$ MANPATH=$MANPATH:/usr/local/pgsql/man
$ export MANPATH
$ man psql
...
```

share

Каталог share содержит примеры файлов конфигурации PostgreSQL и резервных файлов, которые initdb использует для создания начальной базы данных.

data

В каталоге data находится база данных PostgreSQL. Если запустить не один, а несколько процессов, принимающих соединения (postmaster) для разных портов TCP/IP, можно содержать на одном сервере несколько СУБД PostgreSQL. Перечень файлов, входящих в каталог, приведен в табл. 11.2:

Таблица 11.2. Файлы, входящие в каталог data

Файл	Описание
PG_VERSION	Текстовый файл, содержащий версию PostgreSQL данной базы (например, 7.1)
pg_hba.conf	Текстовый файл, содержащий конфигурацию для аутентификации PostgreSQL по имени сервера
pg_ident.conf	Текстовый файл, содержащий конфигурацию для аутентификации PostgreSQL по имени пользователя
postgresql.conf	Файл конфигурации PostgreSQL
postmaster.log	Системный журнал (log file) PostgreSQL (в случае перенаправления из стандартного вывода)
postmaster.opts	Текстовый файл, содержащий параметры командной строки, передаваемые процессу postmaster при его запуске
postmaster.pid	Текстовый файл, содержащий идентификатор процесса postmaster для данной базы. Присутствует, только если сервер работает или произошло аварийное завершение

Сами данные хранятся в подкаталогах global и base. В обычных условиях нет необходимости изучать файлы, находящиеся в этих каталогах, т. к. вся конфигурация осуществляется посредством файлов каталога data.

Инициализация базы данных

После установки PostgreSQL необходимо принять меры по созданию базы данных. В главе 3 это было сделано при помощи `initdb`.

> ***Некоторые дистрибутивы PostgreSQL автоматически вызывают*** `initdb`***, если при запуске компьютера на нем не было баз данных.***

Важно корректно инициализировать базу данных PostgreSQL, т. к. безопасность базы данных связана с распределением прав доступа к каталогам с данными. Для того чтобы обеспечить безопасность базы данных, необходимо выполнить последовательность шагов, приведенных ниже. Достаточно часто сценарий инсталляции пакета PostgreSQL осуществляет эти шаги автоматически. Если требуется изменить значения по умолчанию или если инсталляция проводится вручную, то выполняем их самостоятельно:

- Создаем пользователя, которому будет принадлежать база данных. Рекомендуется назвать его `postgres`
- Создаем каталог (`data`), где будут находиться файлы базы данных
- Обеспечиваем принадлежность этого каталога пользователю `postgres`
- Запускаем `initdb` из-под пользователя `postgres`, а не из-под `root`, для инициализации базы данных

Для утилиты `initdb` можно указать несколько параметров. Самые распространенные из них перечислены в табл. 11.3:

Таблица 11.3. Параметры утилиты initdb

Параметр	Описание
`-D dir` `--pgdata=dir`	Указывает расположение каталога данных для конкретной базы данных
`-i id` `--sysid=id`	Указывает внутренний идентификатор суперпользователя базы данных. Можно опустить, по умолчанию им считается пользователь, запустивший `initdb`
`-W` `--pwprompt`	Программа `initdb` будет запрашивать пароль суперпользователя базы данных. Пароль потребуется для аутентификации по паролю

База данных, создаваемая `initdb` по умолчанию, содержит информацию об учетной записи суперпользователя базы данных (в нашем случае `postgres`) и, наряду с другими данными, базу данных с именем `template1` в качестве шаблона.

Для создания дополнительных баз данных в СУБД PostgreSQL требуется осуществить соединение с СУБД и запросить создание новой базы. Можно сделать это вручную или же применить утилиту `createdb`,

о которой будет рассказано чуть позже. Для соединения необходимы имена пользователя и базы данных. При начальной инсталляции существует только один пользователь, от имени которого можно выполнить соединение, и одна база данных.

Прежде чем рассматривать соединение с СУБД, надо запустить серверный процесс.

Управление сервером

Сервер базы данных PostgreSQL работает в системах UNIX и Linux как процесс, ожидающий соединения (a listener process), а в Windows-системах – как пользовательская программа или системный сервис. Как мы уже знаем из главы 3, серверный процесс называется `postmaster` и должен запускаться, чтобы позволить клиентским приложениям соединиться с базой данных и работать с ней.

При желании процесс `postmaster` можно запускать вручную. Если аргументы не заданы в командной строке, то сервер будет работать с высоким приоритетом, писать сообщения на стандартный вывод и использовать базу данных, хранящуюся там, где указано в переменной окружения `$PGDATA`.

Однако обычно бывает удобнее, чтобы серверный процесс работал как фоновый, а журнал сообщений велся в файле. При осуществлении попытки соединиться с базой данных `postmaster` запускает другой процесс, `postgres`, для обработки доступа к базе данных со стороны подсоединяющегося клиента.

Чтением и изменением данных от имени клиентского приложения занимается сервер. Одновременно может существовать несколько процессов `postgres`, поддерживающих несколько клиентов, но общее количество процессов `postgres` ограничено некоторым максимумом, который определяет `postmaster`. Программа `postmaster` имеет несколько параметров, позволяющих контролировать ее поведение. Наиболее часто используемые параметры представлены в табл. 11.4:

Таблица 11.4. Параметры программы postmaster

Параметр	Описание
`-B nbufs`	Устанавливает количество буферов разделяемой памяти в `nbufs`. Каждый имеет размер 8 Кбайт. Значение по умолчанию – 64 буфера. Задаваемая величина должна как минимум вдвое превышать количество серверных процессов (см. также `N`).
`-d level`	Определяет уровень вывода отладочной информации, записываемой в журнал сервера. Уровень, установленный в `0` (по умолчанию), означает отсутствие отладочной информации, поддерживаются значения вплоть до `4`.

Параметр	Описание
-D dir	Устанавливает каталог базы данных (/data) в dir. Значение по умолчанию не задано. Если параметр -D не указан, то используется значение переменной окружения PGDATA.
-I	Разрешает удаленные TCP/IP-соединения с базой данных. Без этого параметра только клиенты, работающие на машине, где запущен сервер базы данных, могут соединиться с базой. Другие параметры конфигурации, позволяющие обеспечить возможность сетевых соединений, рассмотрены ниже в разделе «Безопасность базы данных» этой главы.
-l	Делает возможными защищенные соединения с базой данных с применением протокола защищенных сокетов (Secure Sockets Layer, SSL). Необходим параметр -i (доступ по сети). Сервер должен быть скомпилирован с поддержкой SSL.
-N cons	Задает максимальное количество одновременных соединений, допускаемых сервером. По умолчанию равен 32. Если значение увеличивается, необходим параметр -B.
-o "opts"	Устанавливает параметры, которые передаются фоновому серверному процессу, обрабатывающему индивидуальные клиенты (postgres). О параметрах серверных процессов читайте в Руководстве администратора PostgreSQL.
-p port	Задает номер TCP-порта, который сервер должен использовать для прослушивания соединений с базой данных. По умолчанию равен значению $PGPORT или указанному при компиляции (обычно 5432).
-c name=value	Устанавливает параметр времени выполнения программы. См. ниже раздел «Конфигурирование сервера в процессе работы».

В случае успешного запуска процесс postmaster создает файл, содержащий идентификатор этого процесса и каталог данных базы данных. По умолчанию это файл:

```
/usr/local/pgsql/data/postmaster.pid
```

Вывод сообщений серверного процесса должен быть перенаправлен при помощи обычного для оболочки перенаправления стандартного вывода и стандартного вывода ошибок:

```
postmaster >postmaster.log 2>&1
```

Как уже говорилось, процесс postmaster должен быть запущен от имени пользователя, отличного от root, созданного как владелец базы данных. Такой пользователь (postgres) создавался в главе 3.

Запуск и остановка сервера

Стандартный дистрибутив PostgreSQL содержит утилиту pg_ctl, управляющую процессом postmaster. Она может запустить, остановить и перезапустить сервер, а также выдать сообщение о статусе сервера:

```
pg_ctl  start  [  -w  ]  [ -D datadir ]  [ -p path ]
[ -o options ]

pg_ctl stop [ -w ]  [ -D datadir ]
[ -m [ s[mart] ]  [ f[ast] ]  [ i[mmediate] ] ]

pg_ctl restart [ -w ]  [ -D datadir ]
[ -m [ s[mart] ]  [ f[ast] ]  [ i[mmediate] ] ]
[ -o options ]

pg_ctl status [ -D datadir ]
```

Чтобы работать с pg_ctl, необходимо иметь право на чтение каталогов базы данных, поэтому надо быть или суперпользователем (root) или пользователем postgres.

Параметры pg_ctl перечислены в табл. 11.5:

Таблица 11.5. Параметры утилиты pg_ctl

Параметр	Описание
-D datadir	Указывает местоположение базы данных. По умолчанию $PGDATA.
-w	Ждать, пока сервер запустится, вместо того чтобы завершиться незамедлительно. Ожидает создания серверного pid (process id — идентификатор процесса) файла. Время ожидания ограничено 60 секундами.
-o "options"	Задает параметры, которые должны быть переданы процессу postmaster при его запуске.
-m mode	Устанавливает режим завершения (smart, fast или immediate).

При остановке или перезапуске сервера задается некоторое количество параметров, относящихся к подсоединенным к базе клиентам:

- По умолчанию pg_ctl stop (или restart) используется с параметром smart (или s), ожидая отсоединения всех клиентов, чтобы остановить сервер.
- fast (f) останавливает базу данных, не ожидая отсоединения клиентов. В этом случае транзакции, находившиеся в процессе выполнения, откатываются.
- immediate (i) отключает базу данных сразу же, не дав серверу возможности сохранить данные. При следующем запуске сервера требуется восстановление данных. Такой режим должен использоваться только при возникновении серьезных проблем.

Можно проверить, работает ли PostgreSQL, используя `pg_ctl status`. Будет выдан идентификатор процесса `postmaster`, ожидающего соединения, и командная строка, посредством которой он был запущен:

```
# pg_ctl status
pg_ctl: postmaster is running (pid: 486)
Command line was:
/usr/local/pgsql/bin/postmaster '-i' '-D' '/usr/local/pgsql/data'

#
```

Пользователи

Каждому запросу к базе данных PostgreSQL ставится в соответствие пользователь. В этом разделе рассмотрено создание, изменение и удаление пользователей. Далее мы также поговорим о группах пользователей и привилегиях, которые управляют доступом к базе данных.

Каждый пользователь, желающий получить доступ к базе данных, должен быть известен PostgreSQL, информация о нем должна храниться во внутренней таблице системы, `pg_user`. Перечень пользователей можно получить, выведя все строки таблицы `pg_user` в `psql`:

```
bpfinal=# SELECT usename, usesysid, usecreatedb FROM pg_user;
usename  | usesysid | usecreatedb
---------+----------+-------------
postgres |       26 | t
neil     |       28 | t
(2 rows)

bpfinal=#
```

У каждого пользователя есть соответствующий ему идентификатор (`usesysid`) и ряд прав, в том числе право на создание баз данных, представленное выше как `usecreatedb`.

Обратите внимание, что имя пользователя PostgreSQL отличается от регистрационного имени пользователя UNIX, Windows или Linux. Имена создаваемых пользователей PostgreSQL обычно совпадают с регистрационными именами пользователей, подключаемых к базе данных, но это не обязательно. Несмотря на то что при соединении с базой данных клиент PostgreSQL по умолчанию представит серверу базы данных регистрационное имя пользователя, можно заменить его, указав имя пользователя, специфичное для PostgreSQL.

Можно задать соответствие регистрационных имен именам пользователей PostgreSQL. Если есть регистрационные имена `neil`, `rick`, `steve` и `gavin` и есть пользователи PostgreSQL `author` и `reviewer`, то можно создать группы пользователей так, чтобы `neil` и `rick` оба подключались к базе данных как PostgreSQL-пользователь `author`, а `gavin` и `steve` – как

PostgreSQL-пользователь reviewer. Вернемся к этому чуть позже, при обсуждении аутентификации.

CREATE USER

В PostgreSQL есть утилита createuser, оказывающая содействие в создании пользователей PostgreSQL:

```
createuser [options...] username
```

Параметры createuser (табл. 11.6) позволяют указать сервер базы данных, для которого создается пользователь, и определить некоторые возможности, такие как создание базы данных:

Таблица 11.6. Параметры утилиты createuser

Параметр	Описание
-h host --host host	Указывает компьютер сервера базы данных. По умолчанию это локальная машина.
-p port --port port	Задает порт. По умолчанию это стандартный порт, на котором принимает соединения утилита postmaster PostgreSQL, 5432.
-q --quiet	Не выводить результат.
-d --createdb	Разрешить данному пользователю создавать базы данных.
-a --adduser	Разрешить данному пользователю создавать новых пользователей.
-P --pwprompt	Приглашение на ввод пароля данного пользователя. Пароль пользователя требуется для аутентификации.
-I --sysid id	Задает идентификатор пользователя – usesysid – для нового пользователя.
-e --echo	Выводит команду, посланную серверу для создания пользователя.

Утилита createuser, выступая в качестве оболочки, при помощи psql выполняет некоторые команды PostgreSQL для создания пользователя. Команду можно увидеть, задав параметр -e или --echo для createuser:

```
$ createuser -e --pwprompt rick
Enter password for user "rick":
Enter it again:
Shall the new user be allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER "rick" WITH  PASSWORD 'xxx' NOCREATEDB NOCREATEUSER
$
```

В основе утилиты лежит команда SQL (а на самом деле DDL) CREATE USER, полная синтаксическая структура которой выглядит следующим образом:

```
CREATE USER username
[ WITH
[ SYSID uid ]
[ PASSWORD 'password' ] ]
[ CREATEDB | NOCREATEDB ]
[ CREATEUSER | NOCREATEUSER ]
[ IN GROUP groupname [, ...] ]
[ VALID UNTIL 'abstime' ]
```

Распределить пользователей по группам и ограничить время, в течение которого данный пользователь является действительным, можно посредством команды CREATE USER (из psql или других клиентов SQL).

DROP USER

Удалить пользователя из системы PostgreSQL можно либо при помощи команды SQL DROP USER name в psql, либо применив утилиту оболочки dropuser:

```
dropuser [options...] username
```

Параметры dropuser включают в себя те же параметры соединения с сервером, что и createuser, а также дополнительный параметр (табл. 11.7):

Таблица 11.7. Параметры утилиты dropuser

Параметр	Описание
-i --interactive	Приглашение на ввод подтверждения перед удалением пользователя

ALTER USER

Можно изменить атрибуты существующего пользователя с помощью команды ALTER USER в psql:

```
ALTER USER username
[ WITH PASSWORD 'password' ]
[ CREATEDB | NOCREATEDB ] [ CREATEUSER | NOCREATEUSER ]
[ VALID UNTIL 'abstime' ]
```

ГРУППЫ

Группы пользователей имеет смысл организовывать в тех случаях, когда требуется, чтобы пользователи имели свои собственные учетные записи, но при этом должна быть возможность трактовать их одинаково, в частности в том, что касается прав доступа к базе данных.

Создать группу пользователей можно при помощи команды SQL CREATE GROUP:

```
CREATE GROUP name
[ WITH
[ SYSID gid ]
[ USER  username [, ...] ] ]
```

Например, чтобы создать группу authors, включив в нее neil и rick, сделаем следующее:

```
bpfinal=# CREATE GROUP authors WITH USER neil, rick;
CREATE GROUP

bpfinal=#
```

pgAdmin предоставляет простой способ управления пользователями и группами, если есть возможность подключения из клиентской программы Microsoft Windows (рис. 11.1):

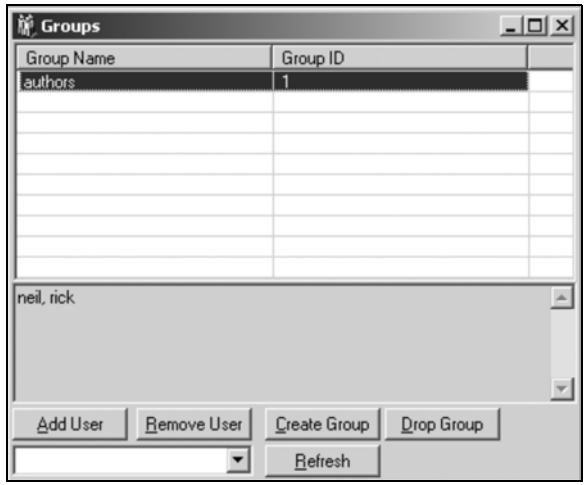


Рис. 11.1. Управление пользователями через клиента Microsoft Windows

Информация о группах пользователей хранится во внутренней системной таблице pg_group, поля которой представлены в табл. 11.8:

Таблица 11.8. Поля таблицы pg_group

Поле	Описание
groname	Имя группы. Избегайте употребления не буквенно-цифровых символов, таких как знаки пунктуации.
grosysid	Числовой идентификатор для группы.
grolist	Список идентификаторов пользователей для членов группы.

Можно добавлять пользователей в существующую группу или удалять их оттуда посредством команды ALTER GROUP:

```
ALTER GROUP name ADD USER username [, ... ]
ALTER GROUP name DROP USER username [, ... ]
```

Для удаления группы пользователей следует выполнить команду DROP GROUP:

```
DROP GROUP name
```

ПРИВИЛЕГИИ

PostgreSQL управляет доступом к базе данных с помощью системы привилегий, которые предоставляются и отменяются командой GRANT. По умолчанию пользователи не имеют права записывать данные в таблицы, созданные не ими.

Команда GRANT имеет такой синтаксис:

```
GRANT privilege [, ...] ON object [, ...]
TO { PUBLIC | GROUP group | username }
```

Поддерживаемые привилегии перечислены в табл.11.9:

Таблица 11.9. Привилегии пользователей

Привилегия	Описание
SELECT	Разрешено чтение строк
INSERT	Разрешено создание новых строк
DELETE	Разрешено удаление строк
UPDATE	Разрешено изменение существующих строк
RULE	Разрешено создание правил для таблицы или представления
ALL	Предоставлены все права

Ключевое слово PUBLIC — это сокращение, обозначающее всех пользователей; object — это имя таблицы, представления или последовательности.

Чтобы разрешить группе authors чтение таблицы customer и добавление новых клиентов, выполним:

```
bpfinal=# GRANT SELECT,INSERT ON customer TO GROUP authors;
CHANGE

bpfinal=#
```

Привилегии отменяются командой REVOKE, очень похожей на GRANT:

```
REVOKE privilege [, ...]
ON object [, ...]
FROM { PUBLIC | GROUP groupname | username }
```

Можно отказать пользователю rick в любом доступе к таблице customer:

```
bpfinal=# REVOKE ALL ON customer FROM rick;
CHANGE

bpfinal=#
```

При этом привилегии группы пользователей остаются в силе. Если, например, группа authors имеет право на доступ к таблице customer, а rick является членом этой группы, то доступ ему будет разрешен. Чтобы завершить изменение полномочий, необходимо удалить пользователя rick из всех групп, имеющих доступ к таблице.

Утилита pgAdmin обеспечивает более удобный для пользователя интерфейс для работы с привилегиями (рис. 11.2):

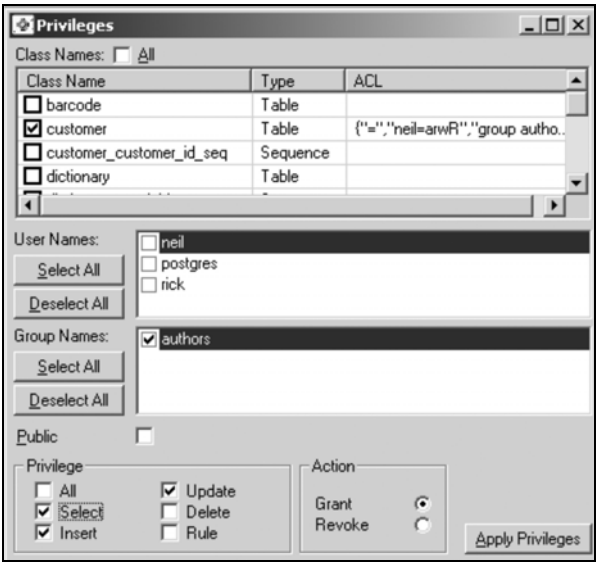


Рис. 11.2. Дружественный интерфейс для работы с привилегиями пользователей

Представления

Комбинируя привилегии и возможности представлений, можно ограничить объем информации из базы данных, доступной для некоторых пользователей. Предположим, что требуется хранить всю информацию о служащих в одной таблице, но ограничить доступ к засекреченной информации, такой как уровень заработной платы, предоставив его только старшим менеджерам. Можно добиться этого, отменив привилегии обычных пользователей, создав представление (см. главу 8), которое выбирало бы только несекретные столбцы, и наделив пользователей правами на доступ к такому представлению.

В PostgreSQL для управления доступом к таблице пользователей служит представление pg_user. «Настоящие» данные о пользователях, в том числе их пароли, хранятся в таблице pg_shadow. У обычных пользователей нет прав на чтение (SELECT) этой таблицы:

```
bpfinal=# SELECT * FROM pg_shadow;
ERROR:  pg_shadow: Permission denied.

bpfinal=#
```

Однако часть таблицы можно прочитать с помощью представления pg_user:

```
bpfinal=# SELECT usename,passwd FROM pg_user;
usename  |  passwd
---------+-----------
postgres | ********
neil     | ********
rick     | ********
(3 rows)

bpfinal=#
```

Столбец passwd возвращается в виде строки звездочек. Остальные же столбцы доступны.

Посмотреть на определение представления pg_user можно посредством внутренней команды psql, \d:

```
bpfinal=# \d pg_user
View "pg_user"
Attribute   |  Type  | Modifier
------------+--------+----------
usename     | name    |
usesysid    | integer |
usecreatedb | boolean |
usetrace    | boolean |
usesuper    | boolean |
usecatupd   | boolean |
passwd      | text    |
valuntil    | abstime |
View definition: SELECT pg_shadow.usename, pg_shadow.usesysid,
pg_shadow.usecreatedb, pg_shadow.usetrace, pg_shadow.usesuper,
pg_shadow.usecatupd, '********'::text AS passwd, pg_shadow.valuntil FROM
pg_shadow;

bpfinal=#
```

В данном случае представление pg_user выбирает все столбцы таблицы pg_shadow, но значение пароля устанавливается в фиксированную текстовую строку.

Сопровождение

Сопровождение является жизненно важной составляющей администрирования баз данных. Рассмотрим следующие его аспекты:

- Создание и удаление баз данных и схем
- Резервное копирование и восстановление данных
- Обновление версии PostgreSQL

Создание и удаление баз данных

Базы данных PostgreSQL создаются в `psql` с помощью команды `CREATE DATABASE`, которая имеет такой синтаксис:

```
CREATE DATABASE name
[ WITH [ LOCATION = 'dbpath' ]
[ TEMPLATE = template ]
[ ENCODING = encoding ] ]
```

Имя базы данных должно быть уникальным в пределах экземпляра PostgreSQL. Местоположение новой базы данных можно указать в параметре `LOCATION`. Подробная информация представлена в документации по PostgreSQL, касающейся `initlocation`. Параметры `TEMPLATE` и `ENCODING` указывают шаблон базы данных и требуемую кодировку. Обычно их можно благополучно не указывать (тонкости применения описаны в документации по PostgreSQL).

> ***Чтобы работать с*** `psql`***, необходимо подключиться к базе данных, поэтому для создания первой базы данных нужно установить соединение с базой данных по умолчанию –*** `template1`**.**

Для удаления базы данных выполните команду `DROP DATABASE`:

```
DROP DATABASE name
```

Нельзя удалить базу данных, выбранную в текущий момент в `psql`. Для удаления базы, с которой установлено соединение, необходимо сначала «переподключиться» к другой базе данных или к `template1`.

PostgreSQL предоставляет две интерфейсных утилиты: `createdb` и `dropdb`, обеспечивающие создание и удаление базы данных соответственно из обычного командного процессора UNIX или Linux:

```
createdb [ опции... ]  dbname [ описание ]
dropdb [ опции... ]  dbname
```

Параметры этих утилит очень похожи на параметры `createuser` и `dropuser`, рассмотренные ранее (табл. 11.10):

Таблица 11.10. Параметры утилит createdb и dropdb

Параметр	Описание
`-h host` `--host host`	Указывает сервер базы данных. По умолчанию это локальная машина.
`-p port` `--port port`	Указывает порт. По умолчанию это стандартный порт PostgreSQL, `5432`.
`-q` `--quiet`	Не выводить результат.
`-e` `--echo`	Вывести команду, посланную серверу для создания пользователя.
`-U name` `--username name`	Указывает имя пользователя для осуществления соединения. По умолчанию это текущий пользователь.
`-W` `--password`	Приглашение на ввод пароля.
`-D dir` `--location dir`	Указывает положение базы данных (только для `createdb`). По умолчанию – каталог инсталляции.
`-E type` `--encoding type`	Указывает тип кодирования базы данных (только для `createdb`). Параметры кодирования описаны в справочном руководстве по PostgreSQL.
`-I` `--interactive`	Запрос подтверждения перед удалением базы данных (только для `dropdb`).

Необязательный параметр `[описание]` для `createdb` позволяет сопоставить новой базе данных комментарий.

Резервное копирование и восстановление данных

Хорошей практикой для всех баз данных PostgreSQL является создание резервных копий. Хранение копии данных в каком-либо удаленном месте обеспечивает защиту от возможной потери данных.

Резервное копирование и восстановление данных слишком часто недооценивают, а это приводит к пагубным последствиям. СУБД зависит от хранимых в ней данных, а потеря данных может произойти по множеству причин начиная с разряда молнии, которая портит жесткий диск, и заканчивая автоматической работой пальцев, удаляющих не те файлы или ошибками программирования, приводящими к разрушению содержимого базы данных. Необходим хорошо продуманный, протестированный и проверенный в работе план резервного копирования и восстановления данных, желательно с автоматизированным процессом копирования. Он поможет свести потерю данных (которая могла бы привести к закрытию предприятия) к незначительному неудобству.

Несмотря на то что для хранения своих данных PostgreSQL использует обычные файлы файловой системы, не рекомендуется доверять

базы данных обычным процедурам резервного копирования. Если в момент копирования файлов PostgreSQL база данных была активна, то нельзя поручиться за целостность сохраненных данных. Теоретически можно отключить сервер базы данных перед копированием файлов, но есть и более совершенный способ. В PostgreSQL реализованы собственные механизмы резервного копирования и восстановления: `pg_dump`, `pg_dumpall` и `pg_restore`.

Проще всего скопировать базу данных, запустив `pg_dump` и перенаправив ее вывод в файл (параметры утилиты `pg_dump` будут представлены чуть позже):

```
$ pg_dump bpfinal > bpfinal.backup
```

По существу, смысл резервного копирования заключается в составлении большого файла команд (сценария) SQL (и внутренних команд PostgreSQL), который при выполнении пересоздаст базу данных в целостности. По умолчанию `pg_dump` выводит текст, и, если внимательно посмотреть на него, можно увидеть операторы для создания пользователей и привилегий, создания таблиц и вставки данных.

Приведем небольшой пример:

```
CREATE SEQUENCE "customer_customer_id_seq" start 1 increment 1 maxvalue
2147483647 minvalue 1  cache 1 ;

--
-- TOC Entry ID 18 (OID 24462)
--
-- Name: customer Type: TABLE Owner: neil
--

CREATE TABLE "customer" (
"customer_id" integer DEFAULT nextval('"customer_customer_id_seq"'::text)
NOT NULL,
"title" character(4),
"fname" character varying(32),
"lname" character varying(32) NOT NULL,
"addressline" character varying(64),
"town" character varying(32),
"zipcode" character(10) NOT NULL,
"phone" character varying(16),
Constraint "customer_pk" Primary Key ("customer_id")
);

--
-- TOC Entry ID 19 (OID 24462)
--
-- Name: customer Type: ACL Owner:
--

REVOKE ALL on "customer" from PUBLIC;
GRANT ALL on "customer" to "neil";
```

```
GRANT INSERT,SELECT on "customer" to GROUP "authors";

--
-- Data for TOC Entry ID 44 (OID 24502) TABLE DATA item
--
-- Disable triggers

UPDATE "pg_class" SET "reltriggers" = 0 WHERE "relname" ~* 'item';
COPY "item"  FROM stdin;
1       Wood Puzzle     15.23   21.95
2       Rubik Cube      7.45    11.49
3       Linux CD        1.99    2.49
4       Tissues 2.11    3.99
5       Picture Frame   7.54    9.95
6       Fan Small       9.23    15.75
7       Fan Large       13.36   19.95
8       Toothbrush      0.75    1.45
9       Roman Coin      2.34    2.45
10      Carrier Bag     0.01    0.00
11      Speakers        19.73   25.32
\.
```

Для восстановления базы данных из резервной копии необходимо выполнить команды из этого файла. Резервная копия содержит команды по созданию и заполнению данными таблиц, но не команду создания базы данных. Предварительно необходимо создать базу данных, в которую будут восстанавливаться данные, и затем уже выполнять команды. Этот метод подходит для копирования базы данных в процессе инсталляции или для ее переименования. Если считать, что создана пустая база данных, например `newbpfinal`, можно восстанавливать данные, используя `psql` для исполнения сценария резервного копирования:

```
$ createdb newbpfinal
$ psql -f bpfinal.backup newbpfinal
```

Параметр `-f` указывает, что `psql` должен читать команды из файла, а не получать их от пользователя.

> ***Если для сопровождения базы данных PostgreSQL применяется программа*** `pgAdmin`***, то она создает таблицы для собственных нужд. Это может привести к ошибкам, о которых будет сообщено при восстановлении данных, поэтому следует либо удалить такие таблицы до выполнения копирования, либо пересоздать их с помощью*** `pgAdmin` ***в новой базе данных до восстановления данных из копии.***

Можно одновременно создать резервную копию всей базы данных (включая внутренние системные таблицы, используемые PostgreSQL), применив `pg_dumpall` от имени суперпользователя базы данных (`postgres`):

```
$ su - postgres
$ pg_dumpall >all.backup
```

Достоинство этого метода заключается в копировании элементов, общих для всех баз данных (например, информации о пользователях). Однако теряется возможность переименования баз.

Для восстановления из копии, содержащей целую базу данных, необходимо сначала подключиться к базе данных по умолчанию `template1`. Полная копия, созданная утилитой `pg_dumpall`, содержит операторы SQL для создания всей базы данных. Вам понадобится выполнить копирование и восстановление как суперпользователю, чтобы иметь соответствующие привилегии на чтение и запись всех данных:

```
$ psql -f all.backup template1
```

Поскольку сценарии резервного копирования – это текстовые файлы, содержащие все данные, а также операторы SQL, то они могут быть очень большими. Можно добиться сжатия копии при помощи программы типа `gzip`:

```
$ pg_dump bpfinal | gzip >bpfinal.backup.gz
$ gunzip -c bpfinal.backup.gz | psql newbpfinal
```

Утилита `pg_dump` имеет множество аргументов/параметров, позволяющих скопировать одну таблицу, указать необходимость включения определений для таблиц и задать формат вывода дампа. Самые полезные параметры перечислены в табл. 11.11.

```
pg_dump [dbname] [options…]
```

Таблица 11.11. Параметры утилиты pg_dump

Параметр	Описание
`-h host`	Сервер, к которому необходимо подключиться. По умолчанию это локальная машина.
`-p port`	Порт TCP/IP, к которому следует подключиться. По умолчанию это стандартный порт PostgreSQL – `5432`.
`-t table`	Указывает, что должен быть выполнен дамп какой-то одной таблицы, а не всей базы данных.
`-u`	Использовать аутентификацию по паролю. Выводит приглашение на ввод имени и пароля.
`-v`	Режим расширенного вывода текстовых сообщений.
`-S user`	Указывает имя суперпользователя базы данных для отключения триггеров.
`-c`	Очистка. Добавляет в сценарий команды удаления таблиц и других объектов перед созданием новых.
`-C`	Создание. Добавляет в сценарий операторы SQL для создания самой базы данных. Только для вывода в текстовом формате.
`-a`	Дамп только данных, без определений объектов.

Параметр	Описание
-s	Дамп только определения объектов (схемы), без данных.
-x	Не выполнять дамп контроля доступа (команды GRANT и REVOKE).
-b	Дамп больших двоичных объектов (BLOB).
-O	Не устанавливать принадлежность объектов для соответствия исходной базе данных.
-f file	Записать сценарий дампа в указанный текстовый файл. По умолчанию – в стандартный вывод.
-F format	Указывает формат вывода дампа. Варианты: p – текстовый сценарий SQL (по умолчанию). t – tar-архив. c – архив заданного пользователем формата.
-Z 0..9	Указывает уровень сжатия для архива указанного формата, от 0 (минимум) до 9 (максимум).

Форматы архивирования вывода дампа (-F t и -F c) делают восстановление базы данных более гибким. Для восстановления с применением архива следует применять утилиту pg_restore:

```
pg_restore [archive] [options...]
```

Наиболее употребительные параметры pg_restore перечислены в табл. 11.12:

Таблица 11.12. Параметры утилиты pg_restore

Параметр	Описание
-h host	Сервер, к которому нужно подключиться. По умолчанию это локальная машина.
-p port	Порт TCP/IP, к которому нужно подключиться. По умолчанию это стандартный порт прослушивания 5432.
-d dbname	Подключиться непосредственно к базе данных dbname, вместо того чтобы использовать psql. Большие двоичные объекты (BLOBs) могут быть восстановлены только так.
-t table	Указывает, что восстановлению подлежит отдельная таблица, а не вся база данных. Возможен лишь один такой параметр.
-P proc	Указывает отдельную функцию (хранимую процедуру), подлежащую восстановлению.
-T trig	Указывает отдельный триггер, подлежащий восстановлению.
-I index	Указывает отдельное определение индекса, подлежащее восстановлению.
-u	Использовать аутентификацию по паролю. Выводит приглашение на ввод имени и пароля.

Таблица 11.12 (продолжение)

Параметр	Описание
-v	Режим расширенного вывода текстовых сообщений.
-l	Выводит содержимое архива. Полезен вместе с -U.
-U file	Восстановить только элементы базы данных, перечисленные в файле. Удобно для выборочного восстановления. Элементы могут быть закомментированы с помощью префиксов --.
-S user	Указывает имя суперпользователя для установления прав собственности и отключения триггеров.
-c	Удалить таблицы и другие объекты перед созданием новых.
-a	Восстановить только данные, без определений объектов.
-s	Восстановить только определения объектов (схему), без данных.
-x	Отменяет восстановление контроля доступа (команды GRANT и REVOKE).
-O	Не устанавливать принадлежность объектов для соответствия исходной базе данных.
-f file	Считывать архив дампа из указанного файла. По умолчанию это стандартный ввод, если файл не указан в первом аргументе.

Для создания резервной копии базы данных и восстановления ее с новым именем можно выполнить такую последовательность команд:

```
$ pg_dump -F c bpsimple >bpsimple.bak
$ createdb bpsimple2
CREATE DATABASE
$ pg_restore -d bpsimple2 bpsimple.bak
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'customer_pk'
for table 'customer'
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'item_pk' for
table 'item'
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'orderinfo_pk'
for table 'orderinfo'
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'stock_pk' for
table 'stock'
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'orderline_pk'
for table 'orderline'
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'barcode_pk' for
table 'barcode'
$
```

База данных bpsimple скопирована в архив формата, определенного пользователем, – bpsimple.bak. Затем создана новая база данных для приема заархивированных данных и их восстановления с помощью утилиты pg_restore.

Обновление версий СУБД

Время от времени выходят новые версии PostgreSQL, содержащие новые возможности или усовершенствования старых, достоинствами которых хотелось бы воспользоваться. Прежде чем пытаться перейти на новую версию PostgreSQL, сделайте копию данных. Лучше обезопасить себя, если нет уверенности в том, что данные могут быть пересозданы, или если они очень важны.

Иногда для перехода на новую версию PostgreSQL требуется выполнить резервное копирование базы данных и восстановить ее. Однако в некоторых ситуациях можно и сохранить данные, переходя на новую версию. Такой способ удобнее при работе с очень большими базами данных, и в некоторых конфигурациях PostgreSQL предоставляет утилиту `pg_upgrade`, с помощью которой можно обновить базу данных. Подробную информацию о ней можно найти на соответствующей странице руководства.

Безопасность базы данных

В большинстве приложений баз данных предъявляются требования к безопасности. Не хотелось бы, чтобы данные были доступны кому угодно, будь то человек или приложение. Идеалом была бы возможность контроля над тем, кто или что получает доступ к данным. Уже говорилось, что в PostgreSQL можно управлять правами доступа пользователей базы данных с помощью команд `GRANT` и `REVOKE` соответственно. Однако этого недостаточно. Необходима уверенность в том, что файлы базы данных защищены от недобросовестного использования. Должна быть возможность определить, что подключающиеся пользователи – действительно те, за кого себя выдают.

PostgreSQL защищает свои базы данных несколькими способами. Во-первых, файлы, в которых PostgreSQL хранит данные, доступны только пользователю PostgreSQL `postgres`. На самом деле суперпользователь операционной системы также может читать файлы данных.

Когда запускается сервер базы данных, он не разрешает удаленные соединения, обслуживая только запросы, сделанные клиентами на локальной машине. Чтобы разрешить сетевой доступ, необходимо явно сконфигурировать PostgreSQL, вызвав `postmaster` с параметром `-i`. Перед этим при желании можно ограничить набор сетевых адресов, соединение которых с сервером разрешено. Другими словами, можно, например, разрешить доступ из локальной сети, но запретить подключение через Интернет. Тогда некто, пытающийся выдать себя за пользователя PostgreSQL, не сможет далеко зайти, и сервер базы данных откажет в подключении, даже не спрашивая о том, кто хочет подключиться.

Установив соединение с сервером, можно сконфигурировать PostgreSQL так, чтобы сделать возможным применение нескольких видов

аутентификации пользователя. Различные методы аутентификации (табл. 11.13) могут быть заданы для отдельных сетевых адресов:

Таблица 11.13. Методы аутентификации

Метод аутентификации	Описание
trust	Соединение всегда разрешено
password	Запрашивается пароль, который сравнивается с записью таблицы pg_shadow
Crypt	Аналогично аутентификации по паролю, только пароль шифруется, а не передается открытым текстом
Ident	Для аутентификации используется сервер ident (RFC 1413)
Krb4	Аутентификация выполняется с использованием Kerberos версии 4
Krb5	Аутентификация выполняется с использованием Kerberos версии 5
reject	В соединении всегда отказывается

Конфигурация защиты PostgreSQL, называемая централизованной, осуществляется посредством файла pg_hba.conf в каталоге данных PostgreSQL. Конфигурационный файл по умолчанию содержит пространные комментарии о структуре записей аутентификации. Сейчас поговорим только об основах аутентификации.

Файл pg_hba.conf состоит из однострочных записей. Пустые строки игнорируются, как и строки, начинающиеся с символа # (они представляют собой комментарии). Каждая запись задает тип аутентификации для какого-то определенного вида соединения. Как минимум, необходимо разрешить подключения с локальной машины или хотя бы с одного сетевого адреса.

Запись для локального соединения имеет вид:

```
local database method [argument]
```

Локальные соединения (осуществляемые через сокеты UNIX-домена, а не сетевые сокеты) с указанной базой данных будут аутентифицироваться с помощью заданного метода. Необязательный аргумент меняется от метода к методу. Для локальных соединений применимы только методы trust, password, crypt и reject.

Приведем пример нескольких локальных аутентификационных записей:

```
# Разрешить всем доступ к тестовой базе данных
local test trust

# Ввести аутентификацию с шифрованием пароля для производственной базы данных
```

```
local bpfinal crypt

# Запретить все остальное
local all reject
```

Осуществляя аутентификацию, PostgreSQL по порядку просматривает такие записи. Применяется первая запись, относящаяся к рассматриваемой базе данных, идущие за ней записи не учитываются. Ключевое слово `all` – сокращение, подразумевающее все базы данных.

Методы `password` и `crypt` поддерживают использование файлов паролей для разных компьютеров сети. Аргументом является имя файла, находящегося в каталоге данных PostgreSQL, предназначенного для хранения паролей, сверка с которыми должна производиться вместо паролей внутренней системной таблицы `pg_shadow`. Такая возможность удобна, если требуется задать различные пароли для одного и того же пользователя, осуществляющего подключения с разных компьютеров.

Сетевые соединения разрешены только клиентским машинам, соответствующим аутентификационной записи сервера (и только если процесс `postmaster` был запущен с параметром `-i`). Сетевая аутентификационная запись имеет вид:

```
host database address netmask method [argument]
```

Дополнительными параметрами являются IP-адрес и маска сети. Любая пытающаяся подключиться машина, сетевой адрес которой соответствует заданному, будет аутентифицирована указанным методом. Представим несколько примеров:

```
# Разрешить подключения из закольцованного сетевого интерфейса
host 127.0.0.1 255.255.255.255 trust

# Разрешить соединения с проверкой пароля из локальной сети
host 192.168.0.0 255.255.255.0 password

# Запретить доступ конкретной машине из другой сети
host 192.168.1.66 255.255.255.255 reject

# Разрешить другим, использующим другой набор паролей
host 192.168.1.0 255.255.255.0 password mypasswords

# Все остальные соединения запрещены
```

Для пользователей, подключающихся из подсети 192.168.1.0, действует другой набор паролей в этой конфигурации. При необходимости можно создать столько файлов паролей, сколько существует машин, подключение которых разрешено.

Файлами паролей занимается утилита `pg_passwd`:

```
pg_passwd password_file
```

Если файл паролей не существует, `pg_passwd` может создать новый файл. Чтобы создать файл `mypasswords`, упомянутый в одном из приме-

ров конфигурации, и задать особый (для определенной машины) пароль для пользователя `neil`, сделаем следующее:

```
$ pg_passwd /usr/local/pgsql/data/mypasswords
File "/usr/local/pgsql/data/mypasswords" does not exist. Create? (y/n): y
Username: neil
New password:
Re-enter new password:
$
```

Подробное рассмотрение других методов аутентификации (таких как `ident` и Kerberos) можно найти в руководстве администратора PostgreSQL или в документации в Интернете.

Параметры конфигурации

PostgreSQL можно сконфигурировать двумя различными способами. Во-первых, во время компиляции PostgreSQL из исходных текстов можно скомпоновать СУБД с модулями, обеспечивающими поддержку некоторых возможностей, и установить некоторые значения по умолчанию.

Во-вторых, после инсталляции PostgreSQL можно задать переменные окружения и параметры командной строки, которые заменят значения по умолчанию или разрешат применение каких-то свойств. Например, можно изменить номер TCP-порта, на котором сервер базы данных (`postmaster`) прослушивает соединения, указав во время компиляции новое значение по умолчанию, используя параметр `--with-pgport=number` для сценария `configure`. Можно заменить значение порта по умолчанию при запуске серверного процесса, используя аргумент `-p number` для `postmaster`.

Конфигурирование сервера в процессе сборки

Во время сборки PostgreSQL из исходных текстов можно задать несколько параметров для сценария `configure` (он использовался в главе 3). Полный перечень параметров можно вывести, запустив `configure` с флагом `--help`:

```
$ ./configure --help
Usage: configure [options] [host]
Options: [defaults in brackets after descriptions]
Configuration:
--cache-file=FILE       cache test results in FILE
--help                  print this message
--no-create             do not create output files
--quiet, --silent       do not print `checking...` messages
--version               print the version of autoconf that created configure
Directory and file names:
```

```
--prefix=PREFIX         install architecture-independent files in PREFIX
[/usr/local/pgsql]
--exec-prefix=EPREFIX   install architecture-dependent files in EPREFIX
[same as prefix]
--bindir=DIR            user executables in DIR [EPREFIX/bin]
...
$
```

Не претендуя на подробное описание параметров, довольствуемся перечислением наиболее часто употребляемых во время сборки, определяющих каталог для установки и некоторые другие свойства (табл. 11.14):

Таблица 11.14. Параметры конфигурирования на этапе сборки

Параметр	Описание
`--prefix`	Установить корневой каталог для установки, обычно `/usr/local/pgsql`
`--with-pgport=port`	Установить номер TCP-порта по умолчанию, обслуживающего сетевые соединения
`--with-maxbackends=n`	Установить максимальное количество одновременных подключений
`--with-tcl`	Включить в сборку модуль поддержки Tcl
`--with-perl`	Включить в сборку модуль поддержки Perl
`--with-odbc`	Включить в сборку модуль поддержки ODBC-соединений

После сборки и инсталляции можно запросить конфигурацию PostgreSQL у `pg_config`:

```
pg_config
--bindir | --includedir | --libdir |
--configure | --version
```

`pg_config` сообщит, в каком каталоге установлены программы PostgreSQL (`--bindir`), укажет расположение включаемых файлов C (`--includedir`) и библиотек объектного кода (`--libdir`), а также версию PostgreSQL (`--version`):

```
# pg_config --version
PostgreSQL 7.1
#
```

Параметры конфигурации, которые использовались в процессе сборки, могут быть получены при помощи команды `pg_config --configure`. Будут выведены параметры командной строки, переданные сценарию `configure`, когда сервер PostgreSQL конфигурировался перед компиляцией.

Конфигурирование сервера в процессе работы

Сервер базы данных PostgreSQL использует некоторое количество параметров, модифицирующих его поведение, причем это может происходить и во время работы. Изменения любого из этих параметров можно добиться одним из трех способов, описанных далее.

При запуске серверный процесс PostgreSQL считывает конфигурационный файл `postgresql.conf` из каталога `data`. Файл по умолчанию является самодокументируемым, в нем содержится запись для каждого параметра, в которой указано значение по умолчанию. В этот конфигурационный файл можно вносить изменения, тогда при каждом запуске сервер будет вести себя соответствующим образом. Есть и другой вариант – можно указать параметр командной строки `-c`, чтобы задать параметр конфигурации при вызове процесса `postmaster`.

Наконец, можно подключиться к действующей базе данных и выполнить команду SQL `SET` для внесения некоторых конфигурационных изменений работающей системы.

Например, одним из параметров конфигурации является уровень отладочных сообщений, записываемых в журнальный файл. Значение по умолчанию равно `0`, но при желании можно установить переменную `debug_level` в `1`, отредактировав `postgresql.conf` так, чтобы он содержал строку:

```
debug_level=1
```

Теперь запустим `postmaster`:

```
postmaster -c debug_level=1
```

Запустите `psql` и выполните команду `set` для установки уровня отладочных сообщений всей системы, а не только текущей сессии `psql`:

```
set debug_level=1;
```

Параметры конфигурации представлены в табл. 11.15:

Таблица 11.15. Параметры конфигурирования в процессе работы

Параметр	Описание
`tcpip_socket`	Разрешить сетевые соединения.
`Max_connections`	Максимальное количество разрешенных одновременных соединений. По умолчанию `32`.
`Port`	TCP-порт для прослушивания соединений.
`Sort_mem`	Объем памяти, используемой для сортировки.
`shared_buffers`	Количество выделяемых буферов разделяемой памяти (как минимум `2 * Max_connections`).
`debug_level`	Уровень отладочных сообщений, записываемых в системный журнал.

Дополнительную информацию о конфигурировании PostgreSQL можно найти во встроенной справке.

Производительность

Последняя остановка нашего путешествия по функциям администрирования PostgreSQL, потенциально являющаяся одной из наиболее значимых (после резервного копирования), – это производительность.

В этой книге работа велась с маленькой базой данных, состоящей из небольшого количества таблиц, каждая из которых содержала несколько строк. Мы не интересовались скоростью, с которой PostgreSQL отвечает на запросы, и физическим размером базы данных. Для реальных баз данных эти вопросы чрезвычайно важны.

Оптимизация баз данных – это мастерство, требующее высокой квалификации в проектировании и досконального знания внутренней работы СУБД. В состав PostgreSQL входит сложный оптимизатор, который пытается выполнять запросы к базе данных как можно более эффективно, но иногда ему требуется помощь.

Не будем стараться охватить все возможности, доступные в PostgreSQL, остановимся лишь на двух достаточно простых вещах, которые помогут поддерживать базы данных PostgreSQL в хорошем состоянии, а именно на:

- Применении команды `VACUUM`
- Создании индексов

VACUUM

В PostgreSQL команда SQL `VACUUM` может использоваться для:

- Очистки дисковой памяти, хранящей базу данных
- Обновления статистических данных оптимизатора

Синтаксис команды таков:

```
vacuum [verbose] analyse [table [ (column [, ... ] ) ] ]
```

С течением времени в таблицах данных PostgreSQL накопится множество недействительных строк, то есть строк, еще занимающих место в базе данных, но уже недоступных. Обычно они появляются в результате отката транзакций.

Если вы помните, в главе 9 говорилось, что во время транзакции, обновляющей строки таблицы, пользователи должны иметь возможность обращаться к таблице с запросами и получать непротиворечивые результаты. PostgreSQL создает новые строки для данных в транзакции и делает их доступными после того, как транзакция зафиксирована. Тем временем запросы возвращают старые строки. После

завершения транзакции в таблице содержатся как старые, так и новые строки, но одно из этих множеств недоступно. Команда VACUUM восстанавливает пространство, занятое такими недоступными строками.

Приведем пример выходных данных команды VACUUM. Добавляем параметр подробного вывода, чтобы увидеть статистические данные. Будем работать с таблицей customer. По умолчанию VACUUM корректирует память для всех таблиц активной базы данных:

```
bpfinal=# vacuum verbose customer;
NOTICE:  --Relation customer--
NOTICE:  Pages 1: Changed 1, reaped 1, Empty 0, New 0; Tup 14: Vac 1, Keep/
VTL
0/0, Crash 0, UnUsed 0, MinLen 120, MaxLen 132; Re-using: Free/Avail. Space
6348/0; EndEmpty/Avail. Pages 0/0. CPU 0.00s/0.00u sec.
NOTICE:  Index customer_pk: Pages 2; Tuples 14: Deleted 1. CPU 0.00s/0.00u
sec.
VACUUM

bpfinal=#
```

Параметр ANALYZE команды VACUUM перерассчитывает различные статистические данные, используемые PostgreSQL для планирования запросов.

Из предыдущих глав мы знаем, что SQL – это декларативный язык. Мы указываем PostgreSQL, какой результат требуется получить (найти всех клиентов, проживающих в Newtown, которые заказали CD с Linux в период между двумя датами), а СУБД решает, как лучше всего это сделать. Как правило, есть несколько вариантов, например можно просматривать таблицу customer, обращаясь к информации о заказах каждого клиента, или же просматривать таблицу item и, найдя изделие «Linux CD», отбирать все содержащие его заказы, отслеживая, какой клиент и когда его разместил.

В зависимости от структуры базы данных, первичных ключей и количества строк в таблицах более быстрым может оказаться или один, или другой способ. PostgreSQL пытается определить, какой способ выполнения запроса будет самым быстрым. Этим и занимается оптимизатор запросов. Перед выполнением запроса он создает его план, учитывающий структуру базы данных, размер таблиц, участвующих в запросе, а также (об этом поговорим чуть позже) доступность индексов для запрашиваемых столбцов.

План для любого конкретного запроса может быть выведен при помощи SQL-оператора EXPLAIN:

```
explain [verbose] query
```

Посмотрим на него в действии:

```
bpfinal=# EXPLAIN SELECT customer_id customer WHERE zipcode='BG3 8GD';
NOTICE:  QUERY PLAN:
```

```
Seq Scan on customer  (cost=0.00..1.18 rows=1 width=4)

EXPLAIN

bpfinal=#
```

В нашей учебной базе данных большинство запросов будут выполнены посредством последовательного просмотра таблиц. PostgreSQL оценивает затраты на осуществление каждой части планируемого запроса и пытается минимизировать итоговую сумму.

В примере, приведенном выше, PostgreSQL оценил стоимость (`cost`) просмотра таблицы `customer` как лежащую в диапазоне от `0` до `1.18`. В данном случае у PostgreSQL нет никакого выбора, можно лишь по очереди проверять все клиентские записи, сравнивая значения `zipcode`.

Оценки «стоимости», выполняемые PostgreSQL, базируются на демографических показателях таблиц, например на количестве строк. Эти показатели не поддерживаются в актуальном состоянии в каждый момент работы сервера, а пересчитываются время от времени.

Это вторая задача, реализуемая командой `VACUUM ANALYZE`:

```
bpfinal=# vacuum analyze;
VACUUM

bpfinal=#
```

PostgreSQL предоставляет утилиту `vacuumdb` для осуществления чистки базы данных из командной строки. Она имеет такой синтаксис:

```
vacuumdb [options] database
```

Параметры `vacuumdb` перечислены в табл. 11.16:

Таблица 11.16. Параметры утилиты vacuumdb

Параметр	Описание
`-h host` `--host host`	Указывает сервер базы данных. По умолчанию это локальная машина.
`-p port` `--port port`	Указывает порт. По умолчанию это стандартный порт PostgreSQL, на котором прослушивается соединение, `5432`.
`-q` `--quiet`	Не выводить результат.
`-U name` `--username name`	Указывает имя пользователя для подключения к базе.
`-W` `--password`	Приглашение на ввод пароля.
`-d name` `--dbname name`	Указывает базу данных, занятая под которую память должна быть восстановлена.
`-a` `-all`	Восстановить память для всех баз данных.

Таблица 11.16 (продолжение)

Параметр	Описание
`-z` `--analyze`	Заново рассчитать статистику для оптимизатора.
`-v` `--verbose`	Вывести подробности процесса очистки, в том числе статистику.
`-t object` `--table object`	Указывает таблицу или столбец, который должен быть почищен. Замечание: объект должен быть заключен в кавычки, если он содержит скобки, обозначающие столбцы.

Настойчиво рекомендуем для базы данных PostgreSQL любого размера запускать `VACUUM` ***или*** `vacuumdb` ***каждый день, например в рамках ежедневного обслуживания. Благодаря этому пространство, занимаемое данными, будет оставаться минимальным, а статистика, используемая оптимизатором, будет актуальной, что обеспечит наилучшую производительность.***

Индексы

Мы уже знаем, что PostgreSQL создает для запроса план, выполненный на основе оценки «стоимости» выборки и просмотра данных. Последовательный просмотр всех строк таблицы по мере увеличения количества строк становится слишком дорогостоящим. Базы данных используют индексы, что позволяет ускорить поиск строк, содержащих определенные данные (т. к. затраты на просмотр индекса обычно значительно меньше, чем на полный перебор).

В действительности PostgreSQL автоматически создает индекс для столбца, определенного как первичный ключ таблицы. Это значит, например, что найти клиента по его `customer_id` можно очень быстро, а вот поиск по почтовому индексу (`zipcode`) потребует полного перебора.

Можно создать дополнительные индексы для таблицы посредством SQL-команды `CREATE INDEX`:

```
CREATE [unique] INDEX indexname ON table(column)
```

Параметр `unique` указывает, что индексируемый столбец не содержит повторяющихся записей, все строки имеют уникальные значения в данном столбце. После создания уникального индекса любая попытка добавить или изменить данные так, чтобы условие уникальности было нарушено, будет пресечена (приведет к ошибке). Используйте этот параметр только в случае полной уверенности в том, что в столбце никогда не появятся повторяющиеся данные.

В учебной базе данных содержится слишком мало данных, чтобы можно было извлечь пользу из применения индексов, поэтому создадим новую таблицу, на которой и продемонстрируем действие индекса.

Чтобы создать большую таблицу, считаем в таблицу словарь UNIX. Получится таблица из более чем 40 000 строк. Можно использовать команду \copy в psql, чтобы считать данные прямо из файла:

```
bpfinal=# CREATE TABLE words ( word text );
CREATE

bpfinal=# \copy words FROM  '/usr/dict/words'
\.

bpfinal=# SELECT count(*) FROM words;
count
-------
45407
(1 row)

bpfinal=#
```

Теперь у нас есть большая таблица. Спросим PostgreSQL, как следует поступить, чтобы найти слово Zulu:

```
bpfinal=# EXPLAIN SELECT * FROM words WHERE word='Zulu';
NOTICE:  QUERY PLAN:

Seq Scan on words  (cost=0.00..22.50 rows=10 width=12)

EXPLAIN

bpfinal=#
```

Несмотря на наличие 45 000 строк PostgreSQL оценивает максимальную стоимость просмотра таблицы в 22,5. На самом деле это просто догадка, весьма далекая от истины. Чтобы помочь PostgreSQL дать более точную оценку, используем VACUUM ANALYZE для обновления статистических данных таблицы после внесения данных:

```
bpfinal=# VACUUM ANALYZE words;
VACUUM

bpfinal=# EXPLAIN SELECT * FROM words WHERE word='Zulu';
NOTICE:  QUERY PLAN:


Seq Scan on words  (cost=0.00..843.59 rows=1 width=12)


EXPLAIN

bpfinal=#
```

Теперь PostgreSQL полагает, что стоимость просмотра таблицы оценивается в 843. Этот пример хорошо иллюстрирует необходимость регулярного запуска VACUUM, обеспечивающего актуальность статистики, особенно после обновления, вставки или удаления значительного объема данных.

Обратившись к базе данным с запросом на поиск Zulu, наблюдаем небольшую паузу (конечно же, на быстрых компьютерах этот запрос будет выполнен с такой скоростью, что вы и моргнуть не успеете):

```
bpfinal=# SELECT * FROM words WHERE word='Zulu';
word
--------
Zulu
(1 row)
bpfinal=#
```

Доступ к таблице words можно ускорить, создав индекс:

```
bpfinal=# CREATE INDEX words_idx ON words(word);
CREATE
bpfinal=#
```

Еще раз посмотрев на план запроса, мы увидим обещанную выгоду. Наблюдается поразительное уменьшение предполагаемых затрат (максимум 2 вместо 843), и запрос теперь выполняется молниеносно:

```
bpfinal=# EXPLAIN SELECT * FROM  words WHERE  word='Zulu';
NOTICE:  QUERY PLAN:
Index Scan using words_idx on words  (cost=0.00..2.09 rows=1 width=12)
EXPLAIN
bpfinal=# SELECT * FROM words WHERE word='Zulu';
word
---------
Zulu
(1 row)
bpfinal=#
```

Можно было ожидать, что индексы помогут только для точных соответствий, но оказывается, что PostgreSQL чуть умнее, чем мы предполагали. СУБД может использовать индекс и для помощи в сопоставлении с префиксом. Если надо найти все слова, которые начинаются с Zu, PostgreSQL использует индекс для поиска частичных совпадений. Приходится осуществлять проверку точного совпадения, но по сравнению с полным перебором строк стоимость такой операции чрезвычайно мала.

Еще раз посмотрим на предполагаемые затраты, выполнив команду EXPLAIN:

```
bpfinal=# EXPLAIN SELECT * FROM words WHERE word LIKE 'Zu%';
NOTICE:  QUERY PLAN:
Index Scan using words_idx on words  (cost=0.00..16.25 rows=14 width=12)
bpfinal=#
```

Увеличение скорости работы базы данных за счет применения индексов может поразить своими масштабами, индексы – это, несомненно,

ключ к максимальному увеличению производительности, но нельзя не упомянуть о том, что они тоже имеют свою цену. Индекс ускоряет доступ, если выборки осуществляются на основе поиска соответствий в индексированном столбце. Но вставки и обновления данных станут более медленными, потому что индекс тоже нужно обновлять. К тому же индексы потребляют пространство, отведенное базе данных.

Необходимо продумывать, для каких таблиц и столбцов базы данных стоит создавать индексы, пытаясь найти золотую середину между возрастанием производительности выборок с одной стороны и увеличением размера базы данных и замедлением обновлений с другой. Итак, какие же таблицы и столбцы индексировать? Жестких правил не существует, и иногда необходимо попробовать, чтобы понять, что и как. Подумайте о том, для чего используется каждая таблица и какие типы запросов могут быть к ней обращены.

Стоит рассмотреть создание индекса для:

- Таблиц, состоящих из большого количества редко обновляющихся строк
- Столбцов, не являющихся первичными или внешними ключами, которые могут использоваться в сложных объединениях
- Столбцов, которые будут просматриваться в поиске точного совпадения или совпадения префиксов

Резюме

В данной главе было рассказано о:

- Некоторых задачах повседневного сопровождения, выполнение которых необходимо для ведения СУБД PostgreSQL
- Способах доступа к серверу базы данных PostgreSQL и контроля над ним
- Управлении пользовательскими учетными записями и контроле за доступом пользователей
- Резервном копировании и восстановлении данных при помощи утилит `pg_dump` и `pg_restore`, которые могут помочь и при обновлении версии PostgreSQL (по мере их выпуска)
- Создании индексов и увеличении производительности

Документация, включенная в состав PostgreSQL, содержит руководство администратора, в котором представлены более сложные темы, не затронутые в главе.

Хотя в этой главе все время использовались утилиты командной строки, можно использовать и `pgAdmin` в качестве графического интерфейса пользователя для более наглядного управления многими свойствами базы данных, в том числе полномочиями пользователей и правами на доступ к таблицам.

12

Проектирование базы данных

До этого момента мы работали с базой данных, содержащей простые сведения о клиентах, заказах и товарах, принимая структуру ее таблиц и столбцов как данность. Теперь же, изучив многие возможности реляционной СУБД, можно вернуться назад и обсудить чрезвычайно важный аспект – проектирование структуры базы данных (или, как ее иногда называют, схемы базы данных).

При написании этой главы мы обратились к коллеге, владеющему искусством проектирования баз данных на самом высоком уровне, с вопросом о том, что он считает самым важным в проектировании. Ответ был прост: «Практика». К сожалению, практику мы не можем заменить ничем, но представим хотя бы основы этого процесса в данной главе. Поговорим о том, как появилась такая схема для учебной базы данных. Также приведем ссылки на некоторые другие книги, чтобы вы могли обратиться к ним и набираться опыта, основываясь на полном понимании задачи.

В этой главе будут рассмотрены следующие аспекты проектирования баз данных:

- Формулирование задачи
- Определение хорошего проекта базы данных.
- Этапы проектирования базы данных
- Проектирование на логическом уровне
- Переход к физической модели
- Нормальные формы
- Общепринятые приемы проектирования

Формулирование задачи

Самым первым шагом на пути к созданию схемы базы данных является осмысление поставленной задачи. Как и при разработке приложений, важно хорошо понимать проблемную область, прежде чем погружаться в тонкости проектирования.

Будет ли проектируемая система использоваться вместо уже существующей? Если да, то вы имеете большое преимущество, поскольку какие бы недостатки и изъяны не присутствовали в старой системе, многие ее свойства наверняка потребуется реализовать и в новой. Даже при таких условиях следует сделать еще одну очень важную вещь: поговорить с потенциальными пользователями системы. Если база данных создается для себя, вопросы все же придется задать, но отвечать на них будете вы сами.

Для того чтобы сделать опрос как можно более продуктивным, проводя его, придерживайтесь следующих принципов:

- Не пытайтесь одновременно опрашивать слишком многих. Достаточно двух-трех человек.
- Заранее оповестите людей о том, что вы хотели бы узнать; если это возможно, разошлите им список основных вопросов.
- Постарайтесь найти помощника, который делал бы заметки, чтобы вы могли сконцентрироваться на восприятии информации, получаемой от пользователей.
- Собрание должно быть недолгим и охватывать разумный объем информации. Если к концу встречи некоторые вопросы или детали остаются нерешенными, можно пригласить кого-то прийти, скажем, через неделю, с подготовленным ответом.
- Обязательно распространите протокол встречи не позднее, чем на следующий рабочий день, с просьбой высказать свое мнение или сделать замечания (если какие-то пункты ставятся под сомнение) в течение недели.

Конкретные вопросы очень сильно зависят от приложения. Однако на первой встрече хорошо начать с таких тем, как описание предназначения системы и ее основных функций. Избегайте вопросов «как?», спрашивайте «что?». Люди часто пытаются рассказать, как все устроено в действующей системе, тогда как вам, чтобы лучше понять положение вещей, требуется знать, почему все так устроено .

Ключ к созданию хорошей схемы в руках потенциальных пользователей, даже если они сами не подозревают об этом. Разрабатывая систему для себя, также стоит потратить время на точное осознание своих целей и потребностей и попытаться предвосхитить возможные изменения.

Хороший проект базы данных

Важно понимать, чего мы пытаемся добиться, проектируя базу данных. В разных системах будут иметь значение различные свойства. Например, база данных может создаваться для хранения результатов некоторого исследования, причем после того, как данные будут извлечены из таблиц, они более не будут использоваться в базе данных. В этом случае обеспечение гибкости для дальнейшего расширения может оказаться не самым разумным занятием.

Рассмотрим те аспекты проектирования, учитывать которые может быть необходимо при создании схемы базы данных.

Способность хранить нужные данные

Объективно ключевым требованием ко всем базам данных является способность хранить нужные данные, т. к. хранение данных – это и есть причина создания базы. Однако выполнение даже этого, казалось бы, универсального требования может быть в разной степени необходимым. Если проектируется достаточно сложная база данных, которая со временем будет развиваться, надо серьезно подумать о том, какие требования «должны быть» выполнены, и реализовать их в первую очередь, а затем уже рассматривать то, что «хорошо было бы» сделать.

Проект базы данных может подвергнуться множеству итераций, напоминая спиралевидную модель разработки приложения, в которой по мере развития системы ее конструкция проходит через множество циклов. Однако есть и существенное отличие, заключающееся в том, что для базы данных гораздо важнее, чем для приложения, чтобы основы схемы были правильными с самого начала. После того как первая итерация базы данных введена в действие и хранит реальные данные, значительные изменения базовой структуры обычно оказываются сложными, отнимающими много времени, к тому же они могут потребовать модификации приложений, работающих с базой данных.

Большая часть схем баз данных в своем конечном варианте будет содержать не более 25% таблиц из выбранного сначала проекта, это так называемые «ключевые» таблицы, составляющие основу схемы. Первоочередной задачей должна быть идентификация и проектирование именно этих центральных таблиц. Остальные таблицы тоже важны, но все же их проектирование играет второстепенную роль.

Способность поддерживать указанные отношения

Схема базы данных должна обеспечивать поддержку отношений между сущностями данных. Если сконцентрироваться только на тонкостях хранения данных, можно не придать значения отношениям между ними, тогда как именно отношения следует считать важнейшим достижением реляционных баз данных. Проект базы данных, в котором собраны все необходимые данные, но проигнорированы отноше-

ния между объектами, будет источником неприятностей, связанных с целостностью данных и излишней сложностью приложений (другие части системы попытаются компенсировать недостатки схемы базы данных).

Способность решить задачу

Базы данных с самой замечательной схемой бесполезны, если они не в состоянии решить проблему, для преодоления которой они создавались. На всем протяжении процесса проектирования необходимо сохранять контакт с проблемной областью. По возможности общайтесь с будущими пользователями, рассказывайте им об основных элементах проекта.

Недостаточно просто отправить пользователям по электронной почте схему базы данных. Нужно посидеть и поговорить с ними, объясняя цели проекта в терминах бизнеса. Делая это, помните о двух ранее обсуждавшихся аспектах проектирования, рассказывайте не только о хранении данных, но и о том, как каждый основной объект может быть связан с другими. Если схема позволит IT-персоналу сопровождать лишь одно подразделение, необходимо упомянуть о такой детали.

Также важно (если возможно) тщательно выбирать пользователей для беседы. Самыми ценными собеседниками являются люди, обладающие самым большим опытом в данном вопросе. К сожалению, они обычно занимают высокие посты и поэтому слишком заняты.

Способность обеспечить целостность данных

Этот аспект тесно связан с предыдущим, касающимся отношений. Цель создания базы данных заключается в хранении данных, и качество последних чрезвычайно важно. В реальном мире данные могут иметь множество недостатков: они могут быть двусмысленными, в заполненных вручную формах записи иногда отсутствуют или их невозможно прочесть. Но дальнейшему ухудшению качества данных в нашей базе не может быть никакого оправдания.

Следует аккуратно подбирать типы данных, задавать ограничения для столбцов и в случае необходимости писать триггерные функции, стараясь реализовать поддержку данных с возможно большей строгостью. Конечно, не помешают здравый смысл и практичность, но не стоит придумывать данные, которых не хватает. Если в базу данных вводятся результаты опроса, лучше зафиксировать тот факт, что ответ неизвестен, чем придумать его.

Обеспечение производительности

В связи с этим сложным аспектом проектирования базы данных часто цитируют Дональда Кнута: «В преждевременной оптимизации первопричина всех бед». Он говорил о приложениях, но это не менее (а может быть, даже более) верно и для проектирования базы данных.

К сожалению, в большой интенсивно используемой базе данных иногда приходится пожертвовать безупречностью схемы для достижения более практических целей – повышения производительности. Однако прежде чем думать о какой-либо оптимизации, следует сначала создать правильную завершенную схему. Часто такие простые вещи, как создание индекса или изменение запроса, могут значительно улучшить производительность, не нарушая при этом основ построения схемы.

Следует избегать попыток внести множество мелких изменений вроде замены типа VARCHAR на CHAR. Обычно они оборачиваются простой потерей времени, возможности схемы обедняются, и она становится противоречивой. Лучше потратить время на профилирование приложения с целью определения «узких мест» (критических элементов, ограничивающих производительность), а потом уже подумать о том, что следовало бы изменить. Но к изменению самой схемы базы данных (вместо изменений, меньше затрагивающих структуру, таких как добавление индекса или перезапись запроса) в любом случае стоит прибегать лишь в самую последнюю очередь.

Способность вносить будущие изменения

Люди, работающие в сфере программного обеспечения, часто удивляются, что многие программные продукты используются гораздо дольше определенного им срока службы. Для баз данных этот эффект еще более заметен, чем для приложений, потому что перенос данных из старой схемы в новую часто оказывается серьезной проблемой. Всегда найдутся доводы в пользу того, что лучше усовершенствовать существующий проект базы данных, чем начинать с нуля, а потом переносить данные.

Может оказаться, что изменения, сделанные в проекте базы данных ради предполагаемого увеличения производительности, усложнили развитие схемы. Как сказал Алан Перли (Alan Perlis) в одной из своих эпиграмм о программировании: «Оптимизация – помеха эволюции» (см. *http://www.cs.yale.edu/homes/perlis-alan/quotes.html*).

Этапы проектирования базы данных

Итак, мы получили некоторое представление о том, чего следует добиваться при создании проекта базы данных. Поговорим теперь о шагах, которые надо предпринять для осуществления этих целей. При обсуждении необходимости осмысления поставленной задачи упоминалось, что проектирование редко бывает чисто технической задачей. Очень важно правильно сформулировать потребности и ожидания пользователей, прежде чем преобразовывать имеющиеся требования в техническую схему.

Сбор информации

Первым этапом проектирования базы данных является сбор информации о том, для чего она создается. Какова главная причина проектирования базы данных? Вы должны суметь определить предназначение базы данных в нескольких предложениях (может быть, это будет всего одна фраза). Если же простого описания не получается, то, вероятно, причина создания базы еще до конца не осознана.

Итак, прежде чем приступать к сбору подробных требований к системе, необходимо четко представлять себе основное задание. Всегда помните о первоначальном, простом определении цели создания базы данных, и если дальше, по мере продвижения вперед, все будет казаться слишком сложным и «распухшим» от излишеств, вернитесь назад. Лишь поняв, что именно вы пытаетесь сделать, можете приступать к расширению и усложнению.

Если новая база данных призвана заменить уже существующую, то в первую очередь следует изучить структуру исходной базы данных: была ли она реляционной или представляла собой плоский файл, или, может быть, электронную таблицу. Даже если в старой системе очень много недостатков, из нее можно извлечь и хорошее и плохое. Вероятно, многое из того, что в ней хранится, понадобится и в новой системе, а ознакомление с существующими образцами данных поможет почувствовать, как будут выглядеть реальные данные. Выясните, что старая система делала хорошо, что – плохо и чего она совсем не делала. Так вы узнаете, что необходимо исправить в новом проекте.

Все, что система должна делать, следует записать, т. к. в процессе написания умственные способности концентрируются. Если планируется создание отчетов, сделайте один пробный и попросите пользователей прокомментировать его.

На этом этапе следует также обдумать отношения, бизнес-правила и обратить внимание на запрашиваемые специальные свойства. Необходимо аккуратно разделить правила на случайные, вроде «мы привыкли так делать», которые могут измениться, и настоящие правила, основанные на фактах, на сущности вещей, которые вряд ли изменятся. Правила первого типа, скорее всего, будут реализованы с помощью триггеров или даже на уровне приложений, чтобы их можно было без труда модифицировать, а вот правила второго типа надо встроить в схему базы данных, обеспечив целостность данных на низком уровне, т. к. это правила фундаментальные и они вряд ли изменятся.

Логическое проектирование

Определение сущностей

Собрав информацию, приступаем к определению **основных сущностей** (ключевых объектов, которые должны присутствовать в базе данных).

В этот момент еще рано беспокоиться о второстепенных объектах. Необходимо выделить объекты, характеризующие проблемную область. В учебной базе данных ключевыми будем считать клиентов, заказы и товары.

Дополнительные подробности, такие как необходимость отслеживать уровень запасов и беспокоиться о штрих-кодах, пока не так важны, на данном этапе не стоит задумываться и об отношениях между объектами.

Если вы считаете, что основные компоненты базы данных выделены, определите (неформально) атрибуты этих компонентов. Например, можно изобразить список основных компонентов и их атрибутов, записанных на обычном языке:

Клиенты и потенциальные клиенты
Имя
Адрес
Номер телефона

Заказы
Заказанные товары
Дата размещения
Дата доставки
Данные об отправке

Информация о товарах
Описание
Цена покупки
Цена продажи
Штрих-коды
Наличный запас

На данный момент Name (имя) не является зарезервированным словом стандарта, но может стать им впоследствии. Сейчас PostgreSQL примет его в качестве идентификатора столбца, но поскольку в будущем оно может стать недопустимым, лучше избегать такого названия. На этом этапе мы не оперируем словарем языка программирования, поэтому в целях начального проектирования продолжим использовать название Name.

Заметьте, что пока мы не думаем ни о том, как хранить адрес, ни о возможных трудностях, таких как наличие нескольких штрих-кодов у одного продукта. Названия атрибутов носят очень общий характер, например Address (адрес) и Shipping information (данные об отправке). Благодаря этому список атрибутов остается достаточно простым и ко-

ротким, во избежание преждевременной концентрации на деталях, из-за которой можно упустить из виду всю общую картину.

На этом этапе некоторые считают полезным сделать краткое описание каждой сущности. В нашей маленькой базе данных это излишне, поскольку компоненты и так очень простые, но в больших базах данных, особенно если приходится иметь дело с более абстрактными понятиями, такая операция может оказаться полезной.

Пример описания для «Информации о товарах» представлен в табл. 12.1:

Таблица 12.1. Описание объекта «Информация о товарах»

Информация о товарах	**Описание**
Описание	Описание изделия (не более 75 символов), включающее информацию о размере
Цена покупки	Цена, уплаченная поставщику за единицу продукции, без учета каких бы то ни было затрат на доставку или налогов
Цена продажи	Цена, которая должна быть заплачена за это изделие, без учета налога с продаж и затрат на транспортировку
Штрих-коды	Штрих-код EAN13
Наличный запас	Количество, имеющееся на складе

Завершив данный этап, остановитесь и проверьте еще раз информацию, собранную на начальной стадии, и убедитесь, что не упустили чего-то важного.

Преобразование сущностей в таблицы

Следующий, более формальный шаг состоит в преобразовании компонентов и списков атрибутов в нечто, напоминающее базу данных. Начнем с поиска разумных имен для таблиц.

Старайтесь по возможности сделать так, чтобы имя таблицы было существительным в единственном числе и состояло из одного слова, даже если получается не вполне естественный результат. В нашей учебной базе данных не составляет труда представить более лаконичные версии названий, такие как `customer` (клиент) или `order` (заказ). Вместо «Информации о товарах» используем `item` (изделие).

Теперь дадим более выразительные имена атрибутам, а также разобьем некоторые наиболее общие описания на столбцы, которые хотелось бы видеть в базе данных. При разделении описаний на столбцы очень важно добиться того, чтобы в каждом столбце хранился ровно один атрибут. Как вы узнаете из продолжения данной главы, это необходимо для приведения базы данных к первой нормальной форме, что является основным требованием к схеме реляционной базы данных.

Начнем с таблицы `customer`:

Customer (клиент)
Name (имя)
Address (адрес)
Phone number (номер телефона)

Name достаточно просто разбить на части. Обычно имени человека предшествует некоторый «титул» (title): Mr, Miss или Dr, поэтому нужен соответствующий столбец. Имена вообще достаточно сложны. Часто люди решают отвести для имени один столбец, предполагая, что в дальнейшем они всегда смогут разбить его на части, если понадобится, например, только фамилия. Это суждение основывается на предположении. Избегайте предположений!

Пусть у некоего клиента двойная фамилия, например «Rose Martin», или немецкая фамилия типа «von Neumann». Кто-то может захотеть указать наряду с фамилией два имени, например «Jennifer Ann Stones». Получится такая таблица с данными:

Title	**Name**
Miss	Jennifer Ann Stones
Dr	John von Neumann
Mr	Andrew Stones
Mr	Adrian Alan Matthew
Mr	Robert Rose Martin

В дальнейшем будет невозможно с уверенностью разделить имена и фамилии. Гораздо удобнее сделать это на начальном этапе и хранить их в базе данных отдельно друг от друга:

Title	**Fname (имя)**	**Lname (фамилия)**
Miss	Jennifer	Stones
Dr	John	von Neumann
Mr	Andrew	Stones
Mr	Adrian	Matthew
Mr	Robert	Rose Martin

Обратите внимание, что было принято решение не хранить «вторые имена» (middle names), а также (что принципиально) хранить только одно имя.

Тогда в будущем мы сможем работать с компонентами имени по отдельности. Например, составляя письмо Dr John von Neumann (а не Dr Neumann), можно будет начать его с обращения Dear John..., а не Dear

John von. Небрежность и неаккуратность производят плохое впечатление на клиентов.

Перейдем к адресу (Address). Обрабатывать в базе данных адреса всегда бывает непросто, т. к. форма записи адреса имеет несколько разновидностей даже внутри одной страны, не говоря уже о разных странах.

Например, в Великобритании адреса записываются в таком виде:

20 James Road,
Great Barr,
Birmingham
M11 2BA

Но в некоторых адресах может вообще не быть номера дома:

Arden House,
Warwick Road,
Acocks Green,
Birmingham
B27 6BH

Американские адреса похожи на английские:

29 S. La Salle St,
Suite 520
Chicago
Illinois
60603

А вот в Германии и Австрии все по-другому (рис. 12.1).

Getreidegasse 9
A-5020 Salzburg

Рис. 12.1. Getreidegasse в Зальцбурге; в доме №9 родился Моцарт

Спроектировать стандартную адресную структуру нелегко, и общепринятого правильного решения не существует. Обычно, как минимум, выделяют город и почтовый индекс (или его эквивалент), как и было сделано в учебной базе данных. В настоящей базе лучше иметь не менее трех строк для адреса плюс город, индекс, штат (для США) и, если это важно, страна.

Если вы живете не в США, то должны были обратить внимание на распространенную ошибку, встречающуюся в веб-формах: предполагается, что в любой адрес входит часть State (штат) и предоставляется удобный список для выбора. Иногда поле делают обязательным для заполнения, но забывают о возможности оставить это поле пустым для оставшейся части планеты. Досадно (людям, живущим вне США) при вводе адреса обнаружить, что обязательно должен быть введен

штат, в то время как такая информация не имеет к ним никакого отношения.

Обычно лучше не настаивать и на введении номера дома, чтобы не создать проблем людям, которые работают в офисах, расположенных в здании, имеющем название, но не номер.

Можно разрешить прием произвольного количества строк адреса, сохраняя их в отдельной таблице. Здесь важно помнить о соблюдении порядка строк, чтобы компоненты адреса располагались правильно. Обычно проектировщики считают, что это избыточно и вполне достаточно разбиения адреса на фиксированное количество строк. В некоторых случаях слишком мелкое дробление может сослужить плохую службу.

Остановившись на упрощенном проекте для адресной информации, получим:

Customer (клиент)
Title (титул)
Fname (имя)
Lname (фамилия)
Addressline (строка адреса)
Town (город)
Zip code (почтовый индекс)
Phone (телефон)

Таблица `Item` (информация о товарах) уже имеет практически готовые описания строк:

Item (товар)
Description (описание)
Buy price (цена покупки)
Sell price (цена продажи)
Barcodes (may be several) (штрих-коды, их может быть несколько)
Stock Quantity (количество на складе)

Заметьте, что решение проблемы существования нескольких штрихкодов для одного товара отложено. Вернемся к этому позже.

С таблицей `orders` все аналогично. Некоторые вопросы также пока не рассматриваются (например, наличие нескольких товаров в одном заказе или возможность заказать несколько единиц товара в одном заказе). Очевидно, что для реализации этой таблицы в реальной базе данных понадобится ее дальнейшее разбиение:

Orders (заказы)
Items ordered (заказанные товары)
Quantity of each item (количество для каждого)
Date placed (дата размещения)
Date delivered (дата доставки)
Shipping information (информация об отправке)

Определение отношений и кардинальности

Сейчас у нас есть список основных объектов, а также в первом приближении список основных атрибутов (может быть, пока еще не полный) для каждого объекта. Теперь пришло время нового важного этапа в проектировании – выделения атрибутов, которые могут встречаться по несколько раз для каждого объекта, и определение типов отношений между объектами. Это понятие часто называют **кардинальностью**.

Некоторые считают, что отношения надо рассматривать даже перед созданием списка атрибутов. Мы же полагаем, что перечисление основных атрибутов помогает осмыслению объектов, поэтому и рекомендуем выполнять этот шаг сначала. Нельзя сказать, что один из путей верный, а другой – нет, поэтому выбирайте тот, который больше вам подходит.

Построение диаграмм отношений

Для осмысления схемы базы данных весьма полезным может оказаться графическое представление структуры данных. В кругах проектировщиков баз данных принято несколько техник и стилей изображения диаграмм. Будем придерживаться общепринятых обозначений, но не забудем, что применяются и другие системы.

На этом этапе будем работать над так называемой **концептуальной моделью**. Пока еще мы не заботимся о тонкостях реализации, посвятив себя логической структуре данных. В концептуальной модели данных таблицы представляются прямоугольниками, а отношения между ними – линиями, символы на концах которых указывают тип отношения. Обозначения, которые будут использоваться, представлены на рис. 12.2:

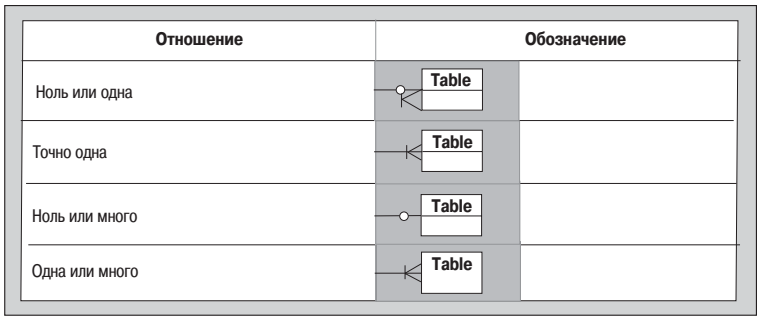


Рис. 12.2. Обозначения, принятые в диаграммах отношений

Отношения между таблицами всегда являются двусторонними, поэтому символ присутствует на каждом конце линии. Читать диаграмму следует по направлению к интересующей вас таблице.

Предположим, что существуют две таблицы, A и B, отношения между которыми изображены на рис. 12.3:

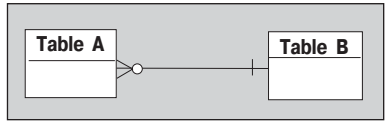


Рис. 12.3. Пример диаграммы отношений

Вот что показано на диаграмме:

- Каждой строке A должна соответствовать ровно одна строка B
- Каждой строке B может соответствовать ноль, одна или несколько строк A

Если бы, например, A была таблицей `orders`, а B – таблицей `customer`, это означало бы, что каждому заказу должен соответствовать ровно один клиент, а у каждого клиента может быть ноль, один или несколько заказов.

Учебная база данных

Освоив азы построения диаграмм отношений, рассмотрим нашу учебную базу данных клиентов, заказов и товаров. В таблице `customer` нет повторяющихся атрибутов, поэтому пока оставим ее в покое. Поработаем над таблицей `item`, особых трудностей здесь возникнуть не должно.

Единственная проблема, связанная с таблицей `item`, заключается в том, что каждый товар может иметь не один штрих-код. Как уже говорилось, в таблице базы данных не может содержаться неизвестное количество повторяющихся столбцов (хотя в PostgreSQL и есть регулярный тип данных, но он не является стандартным и должен использоваться с осторожностью). Предположим, что большинство продуктов имеет два штрих-кода, но есть и продукты, имеющие три, тогда, может быть, самым простым решением будет добавить в таблицу `item` три столбца, `barcode1`, `barcode2`, `barcode3`?

Сначала кажется, что это замечательное решение проблемы, но при более тщательном рассмотрении оказывается, что все не так. Что будет, если вдруг поступит продукт, имеющий четыре штрих-кода? Будем перепроектировать структуру базы данных, добавляя четвертый столбец для штрих-кодов? Скольких столбцов будет «достаточно»? Как говорилось в главе 2, наличие повторяющихся столбцов является очень негибким и почти всегда неверным решением.

Можно подумать и о другом решении – символьной строке переменной длины, в которой будут «спрятаны» штрих-коды, вероятно, разделенные символами, которые не встречаются в штрих-кодах. Это решение также весьма неудачно, т. к. несколько элементов информации хранится в одном и том же месте. Как и в хорошей электронной таблице, очень важно сделать так, чтобы каждый элемент хранился отдельно, чтобы их можно было обрабатывать независимо друг от друга.

Необходимо выделить повторяющуюся информацию, штрих-коды, в новую таблицу. Так удастся разобраться с хранением произвольного количества кодов для каждого продукта. При создании таблицы `barcode` необходимо подумать об отношениях между элементами `item` и `barcode`.

Посмотрим сначала со стороны товаров: каждый товар может не иметь штрих-кода, может иметь один или несколько штрих-кодов. Теперь со стороны штрих-кода: каждый код соответствует ровно одному продукту. Штрих-код всегда является идентификатором самого низкого уровня для товара, идентифицируя различные варианты продукта, такие как рекламные упаковки или упаковки большего объема, в то время как сам продукт не изменяется.

Это отношение показано на рис. 12.4:

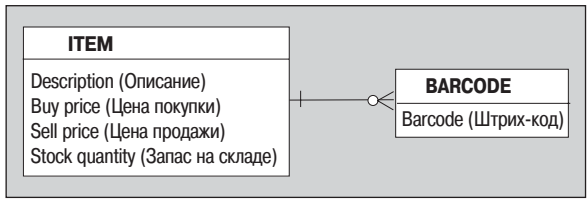


Рис. 12.4. Отношение между таблицами item и barcode

Показано, что продукт может иметь ноль, один или много штрих-кодов, но штрих-код принадлежит ровно одному продукту. Вы должны были заметить, что столбцы для объединения двух таблиц не выделялись. Этим займемся позже. На данном этапе важно определить отношения, а не думать о том, как выразить их в терминах SQL.

Перейдем к таблице `orders`, разобраться с ней будет несколько сложнее. Первый вопрос, на который необходимо ответить, – как представить заказанные товары. В заказах часто содержится несколько продуктов, то есть налицо повторяющаяся информация, относящаяся к заказам. Как и раньше, это означает, что следует выделить товары в отдельную таблицу. Основную таблицу заказов назовем `orderinfo` (информация о заказах), а таблицу, создаваемую для хранения заказанных продуктов, – `orderline` (строка заказа), т. к. можно предположить, что каждая строка таблицы будет соответствовать строке заказа, записанного на бумаге.

Теперь нужно подумать об отношениях между двумя таблицами. Нет смысла в заказе, в котором ничего не заказано, а вот содержать не-

сколько продуктов заказ может, поэтому отношение orderinfo к orderline должно принадлежать к типу «один-ко-многим». Что касается orderline, говорилось, что каждый элемент этой таблицы должен быть связан с ровно одним актуальным заказом, поэтому каждой записи orderline должна соответствовать ровно одна запись orderinfo.

Отобразим отношение на рис. 12.5:

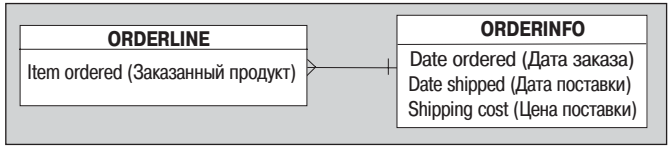


Рис. 12.5. Отношение между таблицами orderline и orderinfo

Если немного подумать, станет очевидным возможное затруднение. Когда люди приходят в магазин, они обычно не просят что-то одно:

```
I'd like a coffee please
I'd like a coffee please
I'd like a donut please
I'd like a coffee please
I'd like a donut please
```

Они просят несколько вещей сразу:

```
I'd like three coffees and two donuts please
```

В настоящий момент наш проект отлично обрабатывает первую ситуацию, но справиться со второй может, только сведя ее к нескольким однострочным.

Можно было бы решить, что это нормально, но если потребуется напечатать заказ на большое количество чашечек кофе, молочных коктейлей и пышек, то клиенту покажется довольно глупым, если каждая позиция будет представлена на отдельной строке. Жизнь усложняется и для нас самих, если, например, необходимо сделать скидку на несколько товаров, заказанных одновременно.

Поэтому лучше хранить для каждой строки количество (рис. 12.6):

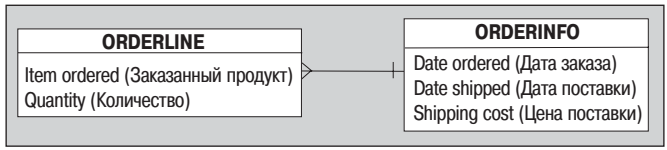


Рис. 12.6. Дополненная таблица orderline

Так можно будет сохранить каждый тип товара, присутствующий в заказе, только один раз, а в отдельном столбце будем хранить требуемое количество продукта.

Теперь, когда базовая концептуальная модель всех объектов готова, свяжем таблицы друг с другом. То, что мы сейчас имеем, показано на рис. 12.7:

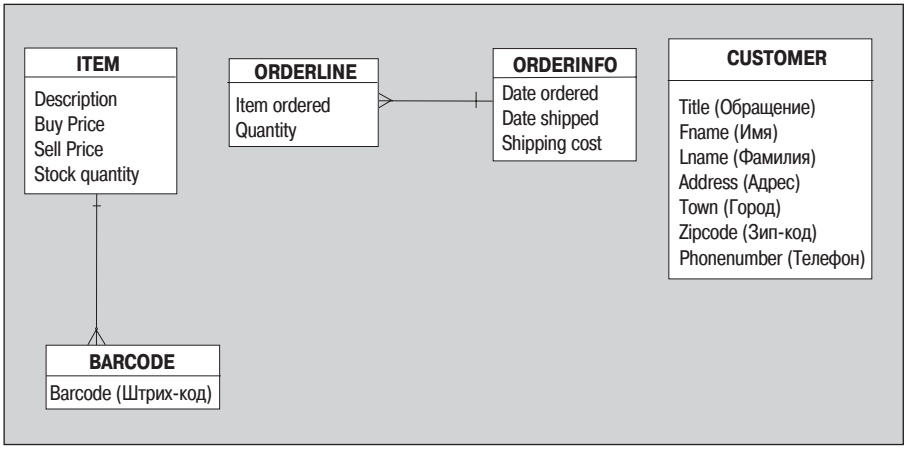


Рис. 12.7. Отношения между всеми имеющимися таблицами на начальной стадии

Подумаем, как же соотносятся между собой три группы таблиц. В такой простой базе данных сразу становится очевидным, что строки `customer` должны быть связаны со строками `orderinfo`. Рассматривая отношения между товарами и заказами, понимаем, что в отношении с таблицей `item` будет участвовать не `orderinfo`, а `orderline`.

Как именно клиенты связаны с заказами? Ясно, что каждый клиент может сделать несколько заказов, при этом каждый заказ сделан ровно одним клиентом, но вот может ли существовать клиент без заказов? Такая ситуация достаточно маловероятна, но все же возможна, например, во время создания учетной записи клиента, поэтому разрешим клиенту не иметь заказов.

Точно так же надо определить отношение между `item` и `orderline`. Каждый элемент `orderline` соответствует ровно одному товару, вот почему в этом направлении отношение имеет тип «один-к-одному». А в противоположном направлении, `item` к `orderline`, каждый отдельный продукт мог еще никогда не заказываться, а мог присутствовать и во множестве заказов, поэтому тип отношения – «ноль или несколько». То, что мы получим, добавив новые отношения на диаграмму, показано на рис. 12.8.

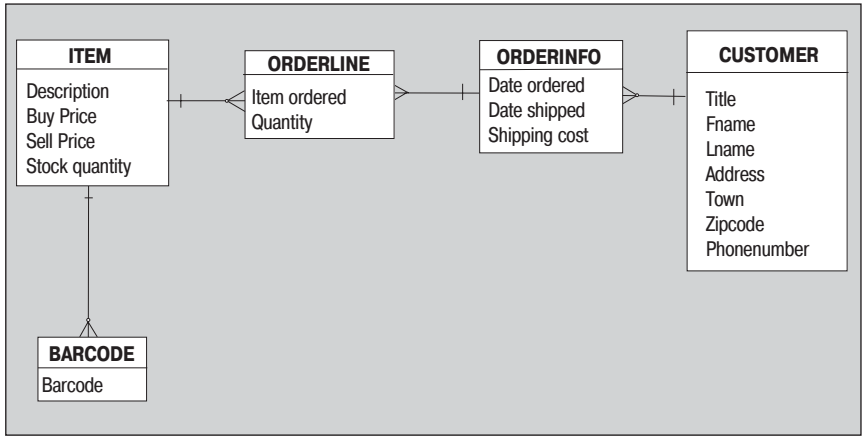


Рис. 12.8. Отображены отношения между всеми таблицами

Итак, на данной стадии у нас есть то, что мы считаем полной схемой основных объектов и их наиболее важных атрибутов, разделенных на части в тех случаях, когда для их хранения необходимо было выделить отдельные столбцы, и диаграмма, показывающая отношения между всеми ними. Первый концептуальный проект нашей базы данных готов!

Теперь жизненно важно сделать паузу и проверить достоверность этого исходного концептуального проекта. Ошибку, сделанную сейчас, исправить позже будет гораздо сложнее. Известный закон программирования гласит: «Чем раньше найдена ошибка, тем легче ее исправить». Некоторые исследования показывают, что с каждым этапом процесса разработки затраты на исправление ошибки возрастают в 10 раз. Не жалейте времени и сил на сбор корректных требований к системе и создание правильного исходного проекта. Это не значит, что нельзя пойти по пути последовательных приближений, если он вам больше нравится. Надо лишь добиться корректности на каждом этапе, правда, изменение схемы несколько сложнее, т. к. после первой итерации в базе данных могут находиться значительные объемы реальных данных и перенос их в новую схему может сам по себе стать непростой задачей.

Если у системы есть будущие пользователи, стоит вернуться к разговору с ними. Покажите им диаграмму и объясните, что на ней изображено, шаг за шагом, чтобы проверить, соответствует ли сделанное их ожиданиям. Если схема частично основана на существовавшей базе данных, вернитесь к оригиналу и убедитесь, что не упустили ничего важного. Большинство пользователей в состоянии понять диаграммы отношений основных объектов базы, если, конечно, вы поясните их. Беседа поможет проверить правильность проекта, к тому же, пользователи почувствуют себя участвующими в разработке.

Переход к физической модели

После проверки логической схемы базы данных на логическую корректность можно начать действовать в направлении получения физического представления этой схемы.

Назначение первичных ключей

Первым шагом обычно является выбор первичных ключей для каждой таблицы. Будем работать над таблицами последовательно, рассматривая их по одной, чтобы решить, какой элемент данных каждой строки делает эту строку уникальной.

Займемся формированием **потенциальных ключей**, элементов данных, которые, вероятно, делают каждую строку однозначно идентифицируемой, а затем выберем один из потенциальных ключей в качестве **первичного ключа**. Если не удается найти ни одного потенциального ключа и если вы считаете их плохими кандидатами в первичные ключи, можно прибегнуть к логическому первичному ключу, который создается специально для того, чтобы действовать в качестве первичного ключа. Но если вам приходится создавать специальные ключи, выполняющие роль первичных, это может свидетельствовать о том, что список атрибутов неполон. Поэтому если очевидные первичные ключи не найдены, рекомендуется пересмотреть список атрибутов.

Сначала будем проверять, не является ли какой-то отдельный столбец уникальным, затем перейдем к изучению комбинаций столбцов. Необходимо также удостовериться, что ни одно из значений ни в одном из столбцов потенциального ключа никогда не может быть неопределенным (`NULL`). Нет смысла в первичном ключе, часть или все значения которого могут быть неизвестны. На самом деле SQL-базы данных, и в том числе PostgreSQL, автоматически вводят ограничение, запрещающее хранить `NULL`-значения в столбце, выступающем в роли первичного ключа.

При поиске столбца, который будет использоваться в качестве первичного ключа, помните о том, что чем меньше длина поля, тем более эффективным будет поиск конкретных значений и тем меньше издержки базы данных. Когда столбец назначается первичным ключом, для него создается индекс. Это делается по двум причинам: чтобы наложить на столбец ограничение уникальности для его значений и чтобы позволить базе данных быстро искать значения в этом столбце. Обычно первичные ключи применяются при поиске гораздо чаще, чем остальные столбцы таблицы, поэтому очень важно обеспечить эффективность такого поиска. Понятно, что просмотр столбца, содержащего 200-символьные описания, будет гораздо более медленным, чем поиск некоторого целого значения.

Если в столбце первичного ключа слишком много символов, то индексное дерево, которое должно быть создано, также будет очень боль-

шим, что увеличит накладные расходы. Поэтому важно постараться выбрать первичными ключами столбцы с короткими полями, идеалом являются целые значения, приемлемы также короткие символьные строки (лучше, если они имеют фиксированную длину). Использования столбцов других типов в качестве первичных ключей лучше избегать.

Таблица Barcode

Начнем с таблицы `barcode`, потому что здесь все будет просто и ясно. Столбец всего один, так что есть только один потенциальный ключ, `barcode`. Штрих-коды уникальны, к тому же обычно они достаточно короткие, вот почему этот потенциальный ключ будет хорошим первичным ключом.

Таблица Customer

Достаточно просто понять, что ни один из столбцов не может однозначно идентифицировать строку, поэтому перейдем к исследованию комбинаций столбцов, которые могли быть первичными ключами. Рассмотрим несколько вариантов:

- Комбинация имени и фамилии. Может быть уникальной, но нет уверенности, что никогда не появится второй клиент с таким же именем.
- Фамилия и почтовый индекс. Эта комбинация лучше, но она также не является гарантированно уникальной, не подходя для случаев, когда (например) муж и жена оба являются клиентами.
- Имя, фамилия и почтовый индекс. Вероятно, уникальность обеспечивается, но все-таки не наверняка. К тому же не очень хорошо получать первичный ключ по трем столбцам. Один гораздо предпочтительнее, хотя можно согласиться и на два.

Итак, явного потенциального ключа нет, поэтому необходимо сформировать логический ключ, который будет уникальным для каждого клиента. Чтобы быть последовательными, всегда будем называть логические ключи следующим образом: `<название_таблицы>_id`, таким образом в данном случае получим `customer_id`.

Таблица Orderinfo

Проблема этой таблицы аналогична таблице `customer`. Явного способа однозначной идентификации строк нет, поэтому опять-таки приходится создавать ключ, на этот раз он будет называться `orderinfo_id`.

Таблица Item

Можно было бы использовать в качестве первичного ключа описание, но дело в том, что описание может быть очень длинной текстовой строкой, а такие строки не могут быть хорошими ключами, поскольку их перебор будет медленным. Поэтому снова создадим ключ – `item_id`.

Таблица Orderline

Таблица `orderline` расположена между таблицами `orderinfo` и `item`. Если считать, что каждый отдельный продукт будет входить в заказ лишь однажды (т. к. продукты, присутствующие в заказе в нескольких экземплярах, обрабатываются при помощи столбца количества), то в качестве потенциального ключа можно рассматривать столбец `item`. На практике же окажется, что это неудачный выбор, т. к. если один и тот же продукт закажут два клиента, то он появится в двух строках `orderline`.

Следует найти какой-то способ связать каждую строку `orderline` с породившим ее заказом `orderinfo`, а т. к. ни один из существующих столбцов не в состоянии обеспечить такую связь, то понятно, что необходимо добавить столбец. Отложим на время вопрос о потенциальных ключах таблицы `orderline` и вернемся к нему чуть позже.

Назначение внешних ключей

Итак, первичные ключи для большей части таблиц выбраны, и пришло время поработать над механизмом, благодаря которому таблицы будут связаны друг с другом. В концептуальной модели содержится информация о том, какие отношения существуют между таблицами, а выбор первичных ключей означает, что найдена возможность однозначной идентификации каждой строки таблицы. При назначении внешних ключей часто бывает так, что необходимо лишь обеспечить наличие столбца, назначенного первичным ключом одной из таблиц, во всех таблицах, непосредственно связанных с исходной.

Заменив имена некоторых столбцов на более содержательные и преобразовав линии отношений в их физический аналог (просто рисуем стрелку, направленную в сторону «должно существовать» таблицы), получим диаграмму, представленную на рис. 12.9:

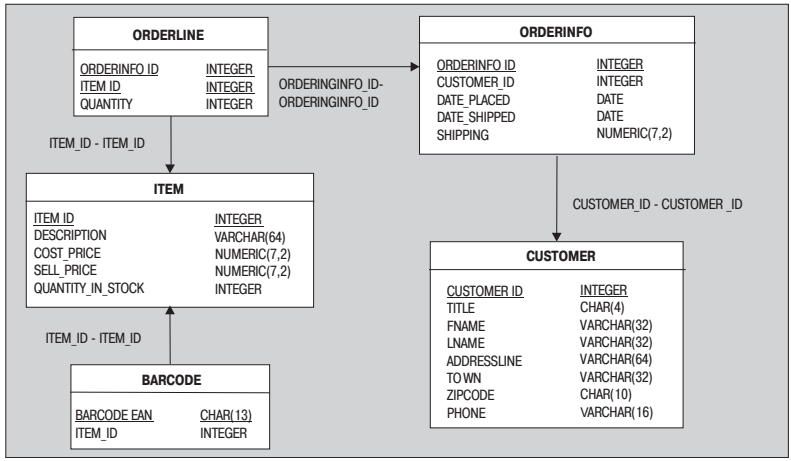


Рис. 12.9. Диаграмма отношений для физической модели базы данных

Обратите внимание, как изменилась диаграмма при переходе от концептуальной модели к физической. Теперь на ней представлены сведения о том, как таблицы могут быть соединены, а не о кардинальности их отношений. Столбцы, являющиеся первичными ключами, на диаграмме выделены подчеркиванием.

Еще рано думать о типах данных и размерах столбцов, займемся этим на следующем этапе. Пока все столбцы сознательно определены как `CHAR(10)`. Вскоре мы вернемся к обсуждению размеров и типов столбцов.

Сейчас надо решить, как связать таблицы друг с другом. Обычно для этого необходимо просто убедиться, что первичный ключ таблицы присутствует и в таблице, связанной с ней. В нашем случае следует добавить столбец `customer_id` в `orderinfo`, столбец `orderinfo_id` в `orderline`, и `item_id` в `barcode`.

Посмотрим теперь на таблицу `orderline` (рис. 12.10):

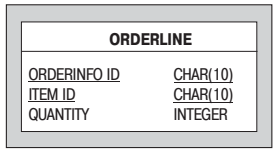


Рис. 12.10. Дополненная таблица orderline

Комбинация `item_id` и `orderinfo_id` всегда остается уникальной. Добавление столбца, необходимого для установления связи между таблицами, решило проблему поиска первичного ключа.

В схеме осталось провести последнюю оптимизацию. Мы знаем, что из-за специфических особенностей нашего бизнеса ассортимент предлагаемых товаров очень велик, но при этом на складе хранятся лишь некоторые продукты. А именно в таблице `item` значение `quantity_in_stock` почти всегда будет равно нулю. Для одного столбца это не так уж важно, но представьте себе, что у нас есть большой объем информации для продуктов, хранящихся на складе, которая всегда остается незаполненной для отсутствующих товаров. Например, можно было бы хранить дату поступления на склад, местоположение склада, сроки годности и номера партий. В целях демонстрации отделим информацию о запасах от информации о собственно товарах и будем хранить ее в специальной таблице.

Теперь физическая модель базы данных с добавленными в нее отношениями и выделенными подчеркиванием первичными ключами выглядит так, как показано на рис. 12.11.

Обратите внимание, что все связанные друг с другом столбцы имеют одинаковые названия. Что на самом деле не обязательно. Столбец `customer_ident` таблицы `orderinfo` вполне может соответствовать столбцу `customer_id` таблицы `customer`. Но работать со схемами баз данных, в которых подчеркнута согласованность, гораздо легче, поэтому (если нет

веских причин для обратного) настойчиво советуем сделать имена столбцов, связанных друг с другом, одинаковыми.

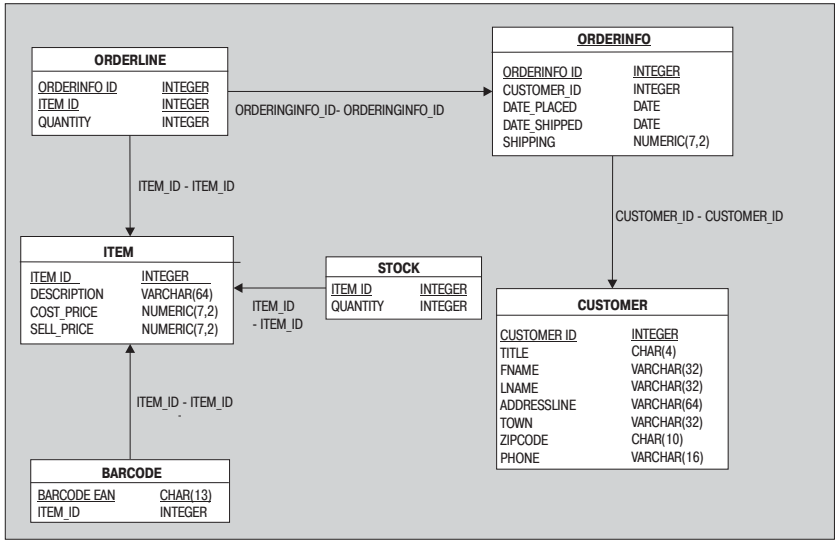


Рис. 12.11. В схему добавлена таблица stock

Вообще, последовательность в именовании очень полезна и удобна. Если требуется создать в качестве первичного ключа таблицы столбец `ident`, придерживайтесь некоторого правила именования, лучше всего придерживаться правила `<название таблицы>_<нечто>`. Не имеет значения, какой суффикс будет выбран: `id`, `ident`, `key` или `pk`, важно, чтобы в присваивании имен прослеживалась последовательность.

Выбор типов данных

Завершив работу по созданию таблиц, добавлению столбцов и установлению отношений, можно перейти к назначению типов данных для всех столбцов. На этом этапе также необходимо выделить столбцы, которые должны принимать `NULL`, и объявить оставшиеся столбцы как `NOT NULL`. Заметьте, начинать следует с предположения, что столбцы должны быть объявлены как `NOT NULL`, и искать исключения. Такой подход лучше, чем изначальное допущение о том, что применение `NULL` допустимо, т. к. (вы уже видели это раньше) обработка неопределенных значений в столбцах может вызвать затруднения, поэтому лучше по возможности минимизировать их вхождения.

Как правило, столбцы, которым отводится роль первичных или внешних ключей, должны принадлежать к машинным типам данных, которые эффективно хранятся и обрабатываются (например, `integer`). PostgreSQL автоматически наложит на такие столбцы ограничение, запрещающее хранение `NULL`.

Часто сложнее всего бывает выбрать тип для хранения денежных единиц. Некоторые предпочитают тип MONEY, если он поддерживается СУБД. В PostgreSQL есть тип MONEY, но руководство пользователя советует применять вместо него числовой тип, что мы и сделаем. Лучше избегать типов с неопределенным порядком округления, таких как тип с плавающей точкой float(P). Типы с фиксированной точностью (numeric(P,S)) гораздо надежнее для работы с финансовой информацией, т. к. задан порядок округления.

Что касается текстовых строк, тут возможностей для выбора очень много. Если длина поля точно известна и является фиксированной, как у штрих-кода, то выбираем CHAR(N), где N – это требуемая длина (предполагаем, что используются только EAN13-коды, хотя на самом деле в употреблении находятся и другие коды, но прибегнем к такому упрощению для иллюстрации примера). Для других коротких текстовых строк также используем тип строк фиксированной длины, например CHAR(4) для «титула». Однако в значительной степени это дело вкуса, и можно выбрать и тип строк переменной длины.

Для текстовых столбцов с переменной длиной в PostgreSQL существует тип text, поддерживающий символьные строки переменной длины. К сожалению, он не является стандартным, и, хотя подобные расширения появляются и в других коммерческих базах данных, в определении ISO/ANSI есть только текстовый тип VARCHAR(N), где N указывает максимальную длину строки. Мы высоко ценим переносимость, поэтому выбираем стандартный тип VARCHAR(N).

Опять же очень важна последовательность. Убедитесь, что все поля типа MONEY имеют одинаковую точность. Проверьте такие универсальные столбцы, как имя и описание, которые могут присутствовать в нескольких таблицах базы данных, необходимо, чтобы они были определены (и, следовательно, использовались) одинаково. Чем меньше в базе данных уникальных типов, тем легче с ней будет работать. Поговорим о назначении типов на примере таблицы customer.

Для начала назначим тип столбцу customer_id. Это первичный ключ (столбец добавлен специально, он станет первичным ключом), так что сделаем его аккуратным и эффективным, выбрав для него тип INTEGER. В столбце «TITLE» (Mr, Mrs или Dr) всегда будет находиться короткая символьная строка, поэтому назначим ему тип CHAR(4). Некоторые проектировщики предпочитают всегда применять тип VARCHAR, чтобы уменьшить количество используемых типов. В данном случае не так уж важно, который из типов выбрать, VARCHAR также вполне подходит. Титул может быть неизвестен, поэтому разрешим хранить в столбце NULL.

Теперь перейдем к столбцам имени и фамилии, fname и lname. Маловероятно, что их длина может превышать 32 символа, но очевидно, что она очень даже переменна, поэтому назначим обоим столбцам тип VAR-

CHAR(32). Столбцу fname разрешим принимать NULL, а вот lname – нет. Клиент с неизвестной фамилией – это выглядит не очень разумно.

В учебной базе данных будем хранить весь адрес целиком, в одном длинном символьном массиве. Как уже говорилось ранее, это может выглядеть как чрезмерное упрощение реальной ситуации, но адреса всегда создают проблемы для баз данных, не существует правильного ответа на вопрос об их хранении.

Дальше продолжаем назначение типов примерно тем же способом. Единственное, на что стоит обратить внимание – номер телефона хранится в виде символьной строки. Хранить телефонные номера в базе как числа почти всегда бывает неправильно, т. к. в этом случае нет возможности хранить международные префиксы, например общепринятая запись кода Великобритании выглядит как +44 (0)116..., где 44 – это код страны, но если вы находитесь в самой Великобритании, то перед кодом зоны нужно набирать еще 0. К тому же в числовом поле не удастся хранить номер, начинающийся с нулей, а в телефонных номерах начальные нули очень даже важны.

Окончательное распределение типов для нашей базы данных представлено на рис. 12.12:

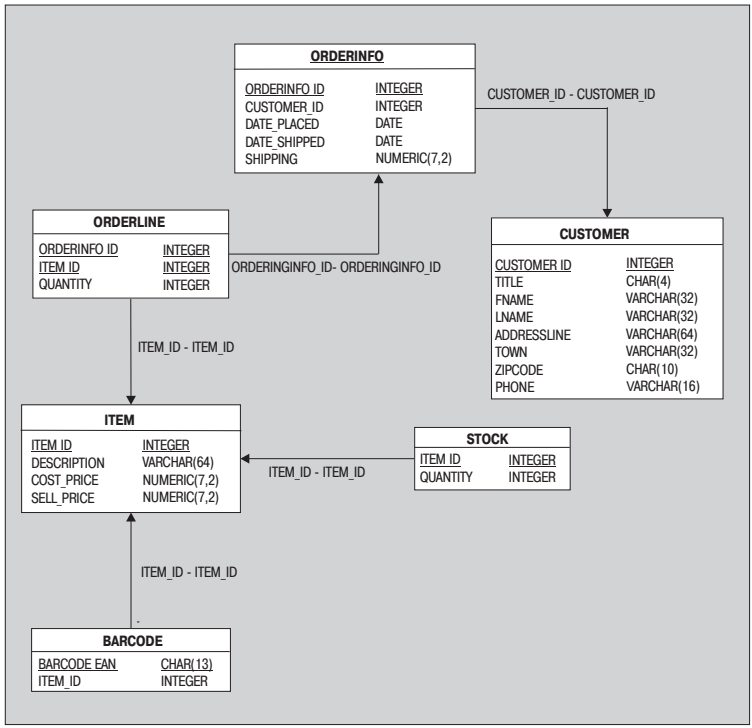


Рис. 12.12. Столбцам назначены типы данных

Завершение определения таблиц

Теперь следует вернуться назад и дважды перепроверить, что в базе данных присутствует вся информация, которую там надо хранить. Должны быть представлены все объекты, а все атрибуты должны быть приведены с соответствующими типами.

На этом этапе можно также решить завести несколько **справочных** таблиц (таблиц со **статическими данными**), например таблицу городов. Как правило, эти справочные таблицы не связаны ни с какими другими таблицами и для приложения просто представляют собою удобный способ гибкого кодирования значений, предлагаемых пользователю. Конечно, можно было бы и жестко запрограммировать эти параметры в приложении, но в большинстве случаев хранение их в базе данных (из которой они могут быть загружены в приложение во время исполнения) значительно облегчает введение дополнительных параметров. Не надо изменять приложение, просто вставляем строки в справочную таблицу базы данных.

Реализация бизнес-правил

Пришло время написать операторы SQL (или воспользоваться утилитой), которые создадут схему базы данных. Можно применить знания, полученные в главах 8 и 10, для того чтобы ввести какие-либо дополнительные бизнес-правила. Про каждое правило стоит подумать, реализовать ли его как ограничение (см. главу 8) или же следует использовать триггер, как в главе 10. Обычно применяются ограничения, потому что с ними легче работать. Несколько примеров ограничений, пригодных к использованию в учебной базе данных, были приведены в главе 10.

Проверка схемы

К этому моменту база данных должна быть готова и дополнена ограничениями (и, возможно, триггерами) для введения в действие бизнес-логики. Прежде чем сдавать сделанную работу и праздновать ее удачное завершение, необходимо еще раз протестировать базу. База данных не является программой в традиционном смысле слова, но это не означает, что ее нельзя протестировать.

Возьмем какие-нибудь данные, если возможно, то пусть это будет часть настоящих данных, которые будут храниться в базе. Вставьте несколько строк в базу данных. Убедитесь, что попытка ввести значение `NULL` в столбцы, где неизвестных величин быть не должно, приводит к ошибке. Попытайтесь удалить данные, связанные с другими данными. Попробуйте нарушить бизнес-правила, реализованные при помощи триггеров или ограничений. Напишите несколько операторов

SQL для объединения таблиц, чтобы сгенерировать данные, подобные тем, которые должны будут присутствовать в отчетах.

После того как база данных сдана в эксплуатацию, становится очень трудно, даже практически невозможно изменить ее схему. Любое, может быть, за исключением самых незначительных, изменение означает остановку системы, выгрузку реальных данных в текстовые файлы, обновление схемы и повторную загрузку данных. Предпринимать такие действия следует только в случае крайней необходимости. Аналогично, если в таблицу загружены неверные данные, часто оказывается, что исправить их или удалить из базы данных очень сложно, т. к. на них могут ссылаться другие данные. Поэтому лучше потратить время на проверку схемы перед развертыванием базы.

Если возможно, вернитесь опять к будущим пользователям и покажите им данные, извлеченные из базы, и как ими можно манипулировать. Даже на этом, позднем этапе из нахождения ошибки, пусть даже самой несущественной, до начала эксплуатации системы можно извлечь значительную выгоду.

Нормальные формы

Глава о проектировании баз данных не может считаться завершенной, если в ней не рассказано о **нормальных формах** и нормализации базы данных. Эта тема была оставлена для конца главы, потому что ее «оторванное от жизни» формальное изложение было бы сухим и малоинтересным, теперь же вы можете посмотреть на то, как соответствует этим сухим правилам схема, через все этапы проектирования которой вы только что прошли.

Считается, что основы нормализации баз данных были заложены в статье Кодда (E. F. Codd), написанной в 1969 году и опубликованной в «Communications of the ACM», vol. 13, № 6 в июне 1970 года. Позднее были определены различные нормальные формы. Каждая нормальная форма основывается на правилах предыдущей и предъявляет более строгие требования к схеме.

Всего существует шесть нормальных форм: первая, вторая и третья нормальные формы, форма Бойса-Кодда и пятая и шестая нормальные формы. Спешим порадовать вас: лишь первые три формы широко используются, поэтому они и будут рассмотрены в этой книге.

Преимущество структурирования данных, соответствующего как минимум трем первым нормальным формам, заключается в том, что с такими данными гораздо легче работать. Плохо нормализованные базы данных трудно сопровождать, и они гораздо более склонны к хранению недействительных данных.

Первая нормальная форма

Первая нормальная форма требует, чтобы каждый атрибут таблицы в дальнейшем не подразделялся и чтобы не было повторяющихся групп. Например, в нашем проекте имя клиента подразделяется на «титул», имя и фамилию. Было известно, что может понадобиться использовать их по отдельности, поэтому они рассматриваются как отдельные атрибуты и хранятся раздельно.

О второй части требования, повторяющихся группах, говорилось в главе 2, когда рассматривалось хранение клиентов и их заказов в простой электронной таблице. Как только клиент размещал более чем один заказ, для него появлялась повторяющаяся информация, и в электронной таблице уже не во всех столбцах содержалось одинаковое количество строк.

Если бы ранее было принято решение хранить два имени клиента в столбце `fname` таблицы `customer`, это нарушало бы первую нормальную форму, т. к. столбец `fname` на самом деле хранил бы «имена», то есть очевидно делимый элемент. В некоторых случаях можно придерживаться практической точки зрения и утверждать (если вы полностью уверены в том, что имена никогда не понадобится рассматривать по отдельности друг от друга), что в терминах схемы базы данных они являются единым объектом. Можно применить и другой, не менее эффективный подход (который мы и выбрали) и всегда хранить только одно имя.

Другим, пугающе распространенным примером нарушения первой нормальной формы, является хранение в одном столбце символьной строки, в которой символы, занимающие разные позиции, имеют разные значения. Так, символы 1–3 обозначают склад, 4–11 – отсек и 12 – полку. Это явное нарушение первой нормальной формы, поскольку подразделы столбца должны рассматриваться по отдельности. На практике также возникают сложности с обработкой, поэтому подобная ситуация должна рассматриваться как недостаток схемы, а не как разумное расширение правил первой нормальной формы.

Вторая нормальная форма

Вторая нормальная форма утверждает, что никакая информация в строке не может зависеть только от части первичного ключа. Предположим, что в таблице `orderline` (рис. 12.13) хранилась бы информация о дате размещения заказа.

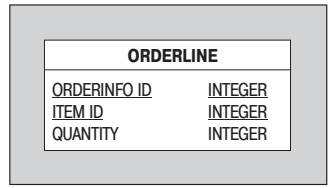


Рис. 12.13. Таблица orderline

Следует помнить, что первичным ключом таблицы orderline является композиция orderinfo_id и item_id. Дата размещения заказа зависит только от данных orderinfo, но не от заказанного товара, вот почему это нарушало бы вторую нормальную форму. Иногда может показаться, что хранимые данные могут нарушить вторую нормальную форму, но на самом деле все оказывается хорошо.

Пусть цены часто меняются. Клиенты справедливо полагают, что они заплатят цену, актуальную на день размещения заказа, а не на дату отгрузки. Чтобы обеспечить это, необходимо хранить в таблице orderline цену, актуальную на день размещения заказа. В этом случае вторая нормальная форма не была бы нарушена, потому что хранимая в таблице orderline цена зависела бы как от изделия, так и от заказа.

Третья нормальная форма

Третья нормальная форма очень похожа на вторую, но имеет более общий характер. Она говорит о том, что данные столбца, не являющегося первичным ключом, не могут зависеть ни от чего, кроме первичного ключа. Третью нормальную форму часто перефразируют таким образом: «Неключевые значения должны зависеть от ключа, от цельного ключа и ни от чего, кроме ключа». Предположим, что в таблице customer (рис. 12.14) хранится также возраст клиентов и дата их рождения.

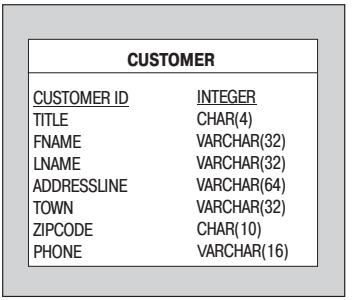


Рис. 12.14. Таблица customer

Это нарушило бы третью нормальную форму, т. к. возраст клиента зависит от его даты рождения, неключевого столбца, как и от конкретного рассматриваемого клиента, задаваемого значением customer_id, т. е. первичным ключом.

Обычно лучше всего привести базу данных к третьей нормальной форме (т. е. сделать так, чтобы ее структура соответствовала требованиям всех трех правил нормализации), но бывают случаи, когда необходимо нарушить правила. Такой процесс носит название денормализации базы данных, порой он необходим для повышения производительности. В любом случае сначала надо спроектировать полностью нормализованную базу, а затем уже заниматься ее денормализацией (если вы считаете, что возникнут серьезные проблемы с быстродействием).

Распространенные приемы проектирования

При проектировании баз данных некоторые вопросы возникают снова и снова, и важно суметь распознать их, т. к. для решения часто встречающихся проблем существуют общие решения. Прежде чем закон-

чить эту главу, вкратце рассмотрим три стандартные проблемы, имеющие не менее стандартные решения.

Отношение «многие-ко-многим»

Похоже, что два объекта связаны отношением «многие-ко-многим». В реальной базе данных недопустимо непосредственно реализовывать такой тип отношений, поэтому необходимо разорвать его.

Решением почти всегда является вставка между двумя исходными таблицами, образующими отношение «многие-ко-многим» дополнительной (связующей) таблицы. Пусть есть две таблицы – `author` (автор) и `book` (книга). Каждый автор мог написать несколько книг, и каждая книга (как, например, та, которую вы читаете) может быть написана несколькими соавторами. Как представить это в физической базе данных?

Решение заключается во вставке между этими двумя таблицами третьей, которая обычно содержит первичные ключи обеих таблиц. В данном конкретном случае создаем таблицу `bookauthor`, которая имеет составной первичный ключ, каждый из компонентов которого представляет собою первичный ключ одной из двух таблиц:

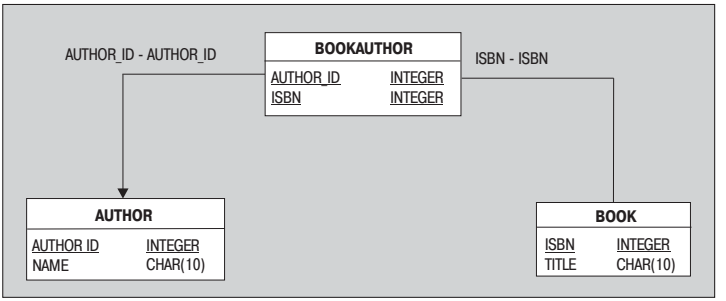


Рис. 12.15. Вставлена связующая таблица bookauthor

Теперь каждый автор встречается в таблице `author` ровно один раз, а вот в таблицу `bookauthor` входит несколько раз, по одному для каждой написанной им книги. Каждая книга ровно один раз присутствует в таблице `book`, но может несколько раз входить в таблицу `bookauthor`, если у нее было несколько авторов. При этом каждая отдельная запись в таблице `bookauthor` остается уникальной, будучи комбинацией книги и автора, которая никогда не повторяется.

Иерархия

Другой часто возникающий вопрос – иерархия данных. Она может скрываться под разными масками. Предположим, у нас есть множество магазинов, каждый из которых расположен в какой-то географической области, области же, в свою очередь, сгруппированы в более крупные объекты, называемые регионами. Допустим, хранение этих

сведений организовано так, как показано на рис. 12.16, т. е. каждый магазин хранит область и регион, в которых он находится:

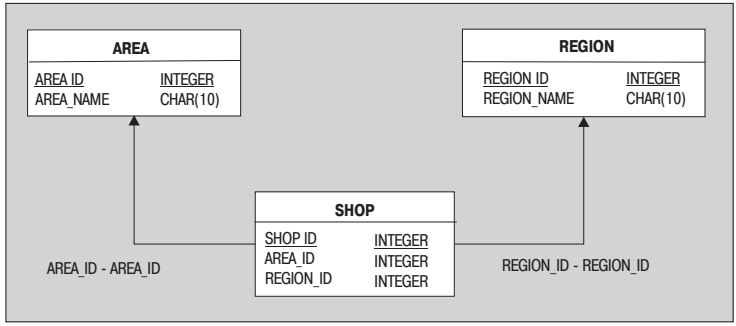


Рис. 12.16. Первая идея

Несмотря на то что это могло бы работать, такой проект далек от идеала, т. к. зная область, мы знаем и регион, поэтому хранение в таблице `shop` и области и региона нарушает правило третьей нормальной формы: регион, хранимый в таблице `shop`, зависит от области, которая не является первичным ключом этой таблицы. Схема, корректно представляющая иерархию (магазин – область – регион), изображена на рис. 12.17.

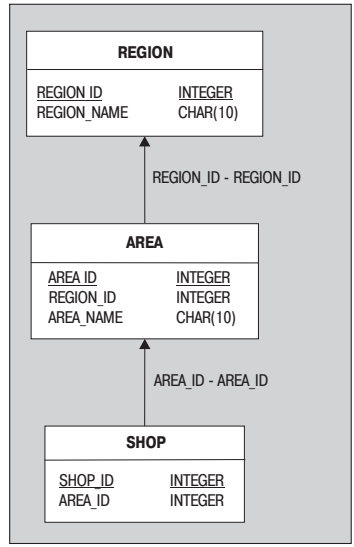


Рис. 12.17. Решение иерархической проблемы

Однако может случиться так, что для повышения производительности придется денормализовать этот идеальный проект и хранить `region_id` в таблице `shop`. В этом случае следует написать триггер, который бы гарантировал, что `region_id`, хранимый в таблице `shop`, всегда корректно указывает на тот же регион, который был бы найден по ссылке через таблицу `area`. Обеспечение уменьшения затрат базы данных на осуществление запросов привело к увеличению расходов на проектирование и усложнению обновлений.

Рекурсивные отношения

Последний описываемый прием не так распространен, как предыдущие два, но часто встречается в двух ситуациях: при представлении иерархической структуры персонала компании и «дроблении номенклатуры» (parts explosion), когда изделия таблицы `item` состоят из других частей, которые, в свою очередь, тоже находятся в этой же таблице.

Рассмотрим пример с персоналом. У всех сотрудников, начиная с самого младшего и заканчивая высшим руководством, есть одинаковые атрибуты, такие как фамилия, табельный номер, зарплата, разряд и адрес, поэтому кажется логичным создать единую таблицу для хранения информации обо всех служащих. Но как тогда сохранить иерархию управления, особенно если разные службы компании имеют разное количество уровней подчиненности?

Ответом является рекурсивное отношение, в котором каждая запись служащего в таблице `person` хранит `manager_id` для указания на руководителя этого человека. Необычность заключается в том, что информация о руководителе хранится в той же самой таблице `person`, порождая рекурсивное отношение. Чтобы найти менеджера для конкретного сотрудника, извлекаем его `manager_id` и ищем его в той же самой таблице, но уже в роли `emp_id`. Сложное отношение с произвольным количеством уровней сохранено в простой однотабличной структуре (рис. 12.18):

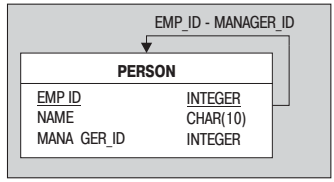


Рис. 12.18. Сложное отношение в простой однотабличной структуре

Предположим, что требуется отразить в базе данных иерархию, представленную на рис. 12.19:

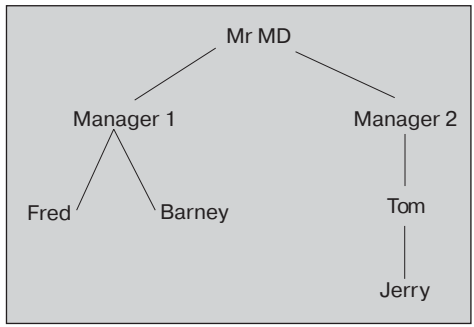


Рис. 12.19. Пример иерархической структуры компании

Строки следует вставлять следующим образом:

```
test=# INSERT INTO person(emp_id, name, manager_id) VALUES(1, 'Mr MD', NULL);
test=# INSERT INTO person(emp_id, name, manager_id) VALUES(2, 'Manager1', 1);
test=# INSERT INTO person(emp_id, name, manager_id) VALUES(3, 'Manager2', 1);
test=# INSERT INTO person(emp_id, name, manager_id) VALUES(4, 'Fred', 2);
test=# INSERT INTO person(emp_id, name, manager_id) VALUES(5, 'Barney', 2);
test=# INSERT INTO person(emp_id, name, manager_id) VALUES(6, 'Tom', 3);
test=# INSERT INTO person(emp_id, name, manager_id) VALUES(7, 'Jerry', 6);
```

Обратите внимание, что первый номер – emp_id – является уникальным, а вот второй номер – это emp_id следующего руководителя в иерархической структуре. Например, сотрудник Tom имеет emp_id, равный 6, а его manager_id равен 3, то есть значению emp_id для Manager2, т. к. Manager2 – это начальник для Tom.

Все хорошо до тех пор, пока не приходит время извлечь данные из этой иерархии. Необходимо выполнить операцию объединения таблицы с самой собой, **самообъединение**. С этой целью требуется создать псевдоним для таблицы (обсуждалось ранее) и написать такой оператор SQL:

```
test=# SELECT n1.name AS "Manager", n2.name AS "Subordinate" FROM person n1,
test-# person n2 WHERE n1.emp_id = n2.manager_id;
```

Созданы два альтернативных названия для таблицы person: n1 и n2, и теперь можно соединять столбец emp_id со столбцом manager_id. Также, при помощи AS, столбцам даны названия, чтобы сделать вывод более содержательным.

В итоге получаем полное представление иерархической структуры таблицы person:

```
  Manager   | Subordinate
------------+-------------
 Mr MD      | Manager1
 Mr MD      | Manager2
 Manager1   | Fred
 Manager1   | Barney
 Manager2   | Tom
 Tom        | Jerry
(6 rows)
```

Ресурсы

Отметьте на будущее две хорошие книги, в которых рассказывается о проектировании баз данных:

- «Database Design for Mere Mortals» (Проектирование баз данных для простых смертных) Майкл Д. Хернандес (Michael J. Hernandez), Addison-Wesley (ISBN 0-201-69471-9). Книга охватывает получение информации, необходимой для проектирования, ее документирование и собственно проектирование базы данных, которое описано гораздо более подробно, чем позволяет размер нашей книги.
- «The Practical SQL Handbook» (Практическое руководство по SQL) Джудит С. Боуман (Judith S. Bowman), Сандра Л. Эмерсон (Sandra L. Emerson) и Марси Дарновски (Marcy Darnovsky), Addison-Wesley (ISBN 0-201-44787-8). В этой книге есть небольшой, но хорошо написанный раздел о проектировании и нормализации баз данных.

Резюме

В данной главе представлен краткий рассказ обо всех этапах проектирования базы данных, начиная со сбора требований, через создание концептуальной схемы к финальному преобразованию этой концептуальной схемы в физический проект базы данных. По ходу повествования был рассмотрен выбор потенциальных ключей, первичных и внешних ключей.

Также описано назначение типов данных для столбцов и указано на важность соблюдения постоянства при проектировании.

Коротко представлены нормальные формы, являющиеся основой хорошей схемы реляционной базы данных. В завершение было рассказано о трех часто возникающих при проектировании базы данных проблемах и о стандартных способах их решения.

13

Доступ к PostgreSQL из C при помощи libpq

В этой главе рассматриваются способы создания собственных клиентских приложений для PostgreSQL. До этого момента мы имели дело либо с приложениями командной строки, такими как `psql`, входящими в дистрибутив PostgreSQL, либо с графическими средствами (например `pgAdmin`), разработанными специально для PostgreSQL. Стандартные средства, подобные Microsoft Access и Sun StarOffice, также можно применять для просмотра и обновления данных по ODBC-соединениям и для создания приложений. Но если необходим полный контроль над клиентскими приложениями, стоит подумать о создании своих собственных. Тут-то на арену и выходит `libpq`.

Вспомните, PostgreSQL имеет архитектуру «клиент-сервер». Клиентские программы, такие как `psql` и `pgAdmin`, могут работать на одной машине, например настольном компьютере с ОС Windows, а сам сервер PostgreSQL работает на UNIX- или Linux-сервере. Клиентские программы по сети посылают запросы к серверу. Эти сообщения абсолютно такие же, как `SELECT` или другие операторы SQL, с которыми мы работали в `psql`. Сервер отсылает обратно множество результатов, которые отображаются у клиента.

Сообщения, которыми обмениваются клиентские приложения и сервер PostgreSQL, форматируются и передаются согласно определенному протоколу. Этот «клиент-серверный» протокол (официального названия не существует, но иногда его называют frontend/backend-протоколом) гарантирует, что в случае потери сообщений будут предприняты адекватные действия, и обеспечивает доставку полного результата. Он также (до определенного уровня) может справляться с несогласо-

ванностью версий клиента и сервера. Клиентские приложения, разработанные для работы с PostgreSQL версии 6.4 или выше, должны взаимодействовать с более поздними версиями без особых проблем.

Утилиты для отсылки и получения таких сообщений входят в библиотеку `libpq`. Чтобы написать клиентские приложение, надо просто использовать эти утилиты и связать наше приложение с библиотекой. Будем действовать в предположении, что читатели обладают некоторыми знаниями языка программирования C.

Функции, обеспечиваемые библиотекой `libpq`, распадаются на три разных категории:

- Соединение с базой данных и управление соединением
- Выполнение операторов SQL
- Чтение результирующих множеств, полученных от запросов

Как и в большинстве продуктов, которые росли и расширялись от версии к версии, в `libpq` часто существует несколько способов сделать что-либо. Основное внимание уделим наиболее распространенным способам, в тех же случаях, когда полезным может оказаться применение альтернативного способа, будут даны соответствующие рекомендации.

Использование библиотеки libpq

Все клиентские приложения PostgreSQL, работающие с библиотекой `libpq`, должны включать в себя соответствующий заголовочный файл, определяющий функции, предоставляемые `libpq`, и компоноваться с соответствующей библиотекой, которая содержит код для этих функций.

Клиентские приложения, известные еще как приложения **frontend** (`fe`), должны включать в себя заголовочный файл `libpq-fe.h`. Этот заголовочный файл предоставляет определения функций `libpq` и скрывает внутреннюю деятельность PostgreSQL, которая может меняться от версии к версии. Заголовочный файл `libpq-int.h`, также входящий в состав дистрибутива PostgreSQL, содержит определения внутренних структур, используемых `libpq`, но его применение в обычных клиентских приложениях не рекомендуется.

Работа с `libpq-fe.h` гарантирует, что ваши программы будут компилироваться с более поздними версиями `libpq`. Заголовочные файлы устанавливаются в подкаталог `include` каталога установки PostgreSQL (по умолчанию `/usr/local/pgsql/include`). Необходимо указать этот каталог компилятору C, чтобы он смог найти заголовочные файлы, с помощью параметра `-I`.

Библиотека `libpq` будет размещена в подкаталоге `lib` каталога установки PostgreSQL (по умолчанию `/usr/local/pgsql/lib`). Чтобы включить функции `libpq` в свое приложение, надо скомпоновать его с этой библио-

текой. Проще всего добиться этого, сказав компилятору, что компоновка осуществляется с `-lpq`, и указав библиотечный каталог PostgreSQL в качестве места для поиска библиотек (при помощи параметра `-L`).

Типичная программа `libpq` имеет такую структуру:

```
#include <libpq-fe.h>

main()
{
    /* Подключиться к базе данных PostgreSQL */

    LOOP:
    /* Выполнить операторы SQL */
        /* Прочитать результаты запроса */

    /* Отсоединиться от базы */
}
```

Программа должна быть скомпилирована и скомпонована в исполняемую программу посредством команды, подобной представленной ниже:

```
$ gcc -o program program.c -I/usr/local/pgsql/include -L/usr/local/pgsql/lib
-lpq
```

В дистрибутиве Red Hat Linux библиотека `libpq` установлена в том каталоге, который компилятор просматривает по умолчанию, поэтому необходимо указать только параметр, определяющий каталог для заголовочных файлов:

```
$ gcc -o program program.c -I/usr/local/pgsql/include -lpq
```

Другие дистрибутивы Linux могут помещать заголовочные файлы и библиотеки в разные каталоги. В Microsoft Windows при использовании пакета Cygwin заголовочные файлы помещаются в каталог `/usr/include/postgresql`, а библиотеки — в `/usr/lib`.

Далее рассказывается, как можно несколько упростить сборку приложения PostgreSQL с помощью Makefile.

Соединение с базой данных

Как правило, клиентское приложение PostgreSQL может во время своей работы соединяться с одной или несколькими базами данных. На самом деле при необходимости можно одновременно подключиться к многим базам данным, управляемым несколькими различными серверами. Библиотека `libpq` предоставляет функции для создания и обслуживания таких соединений.

При соединении с базой данных PostgreSQL на сервере `libpq` возвращает дескриптор этого соединения. Он представлен внутренней структурой, определенной в заголовочном файле как `PGconn`, можете считать

эту структуру аналогом файлового дескриптора. Многим функциям libpq необходим в качестве аргумента указатель на PGconn для идентификации соединения, с которым надо работать (подобно тому, как стандартная библиотека ввода/вывода в C использует указатель на FILE).

Создаем новое соединение с базой данных при помощи PQconnectdb:

```
PGconn *PQconnectdb(const char *conninfo);
```

Функция Pqconnnectdb() возвращает указатель на дескриптор нового соединения. Если по какой-то причине новый дескриптор не сможет быть назначен, например, если не хватит памяти, то функция возвращает NULL. Однако если возвращен не NULL, то это еще не означает, что соединение прошло успешно.

Единственным аргументом PQconnectdb является строка, определяющая, с какой базой данных производится соединение. В нее встроены различные параметры, с помощью которых можно модифицировать способ осуществления соединения. Строковый аргумент conninfo состоит из разделенных пробелами параметров, указываемых в виде параметр=значение. Наиболее употребительные параметры и их описания приведены в табл. 13.1:

Таблица 13.1. Параметры функции PQconnnectdb

Параметр	Описание	Значение по умолчанию
dbname	База данных, с которой осуществляется соединение	$PGDATABASE
user	Имя пользователя, задаваемое при соединении	$PGUSER
password	Пароль для заданного пользователя	$PGPASSWORD или ничего
host	Имя сервера, с которым осуществляется соединение	$PGHOST или localhost
hostaddr	IP-адрес сервера	$PGHOSTADDR
port	Порт TCP/IP для соединения с сервером	$PGPORT или 5432

Для того чтобы соединиться с базой данных bpsimple на локальном компьютере, используем строку conninfo следующим образом:

```
"dbname=bpsimple"
```

Включить в значение параметра пробелы или ввести пустое значение можно, заключив их (его) в одинарные кавычки:

```
"host=monster password='' user=rick"
```

Параметр host задает имя сервера, к которому надо подключиться. Вызов PQconnectdb требует разрешения имени для определения IP-адреса сервера, что позволит осуществить соединение. Обычно этим занимается служба доменных имен (Domain Naming Service, DNS), для завер-

шения может потребоваться немного времени. Если известен IP-адрес сервера, то можно в параметре hostaddr указать адрес, тогда удастся избежать задержки, вызываемой поиском имени. Значения hostaddr указываются в виде четырех чисел, разделенных точками (формат **dotted quad**), что является обычным способом записи IP-адресов:

```
"hostaddr=192.168.0.22 dbname=neil"
```

Если не указан ни host, ни hostaddr, то PQconnectdb будет пытаться соединиться с локальной машиной.

По умолчанию сервер PostgreSQL прослушивает TCP-порт 5432 в ожидании клиентских соединений. Для того чтобы соединиться с сервером, который слушает другой порт, укажите номер с помощью параметра port.

Параметры также можно задать при помощи переменных окружения (см. табл. 13.1). Например, если в аргументе conninfo не задан параметр host, то PQconnectdb обратится к окружению, чтобы посмотреть, задана ли переменная PGHOST. Если да, то значение $PGHOST будет использовано в качестве имени сервера для соединения. Можно запрограммировать клиентскую программу так, чтобы она вызывала PQconnectdb с пустой строкой, а все параметры задавались переменными окружения:

```
#include <libpq-fe.h>

int main()
{
    PGconn *conn = PQconnectdb("");
    …
}
```

```
$ PGHOST=monster PGUSER=neil ./program
```

Как уже говорилось, возвращение функцией PQconnectdb дескриптора соединения, не равного NULL, не гарантирует безошибочного выполнения соединения.

Для проверки состояния соединения надо обратиться к другой функции, а именно к PQstatus:

```
ConnStatusType PQstatus(const PGconn *conn);
```

ConnStatusType — это перечислимый тип данных, включающий следующие константы:

```
CONNECTION_OK
CONNECTION_BAD
```

В зависимости о того, успешно ли прошло соединение, статус дескриптора, возвращенного PQconnectdb, будет равен одному из этих двух зна-

чений. В ConnStatusType входят и другие значения статусов, используемые для других видов соединений.

Зкнчив работу с соединением, его следует закрыть (как и дескрипторы открытых файлов). Делаем это, передавая указатель дескриптора соединения функции PQfinish:

```
void PQfinish(PGconn *conn);
```

Вызов PQfinish позволяет библиотеке libpq освободить ресурсы, которые были заняты соединением.

Теперь можно написать, вероятно, самую короткую полезную программу PostgreSQL (connect.c), позволяющую проверить, можно ли установить соединение с базой данных. Параметры PQconnectdb будем передавать через переменные окружения (хотя это можно делать и при помощи аргументов командной строки и можно даже задать в виде констант в самой программе, если так удобнее для конкретного приложения):

```
#include <stdlib.h>
#include <libpq-fe.h>

int main()
{
  PGconn *myconnection = PQconnectdb("");
  if(PQstatus(myconnection) == CONNECTION_OK)
    printf("connection made\n");
  else
    printf("connection failed\n");
  PQfinish(myconnection);
  return EXIT_SUCCESS;
}
```

```
$ gcc -o connect connect.c -lpq
$ ./connect
connection failed
$ PGDATABASE=bpsimple PGUSER=neil ./connect
connection made
$
```

Makefile

В предыдущей программе были опущены параметры -L и -I для компилятора, это было сделано для удобочитаемости. Всегда помните, что данные параметры необходимы для компиляции программ с использованием libpq. Если вы компилируете с помощью Makefile, то можете добавить их в переменные CFLAGS и LDLIBS соответственно.

Представим очень простой Makefile, который может быть использован для компиляции всех программ, приведенных в качестве примеров

в этой главе. Его и исходные тексты примеров можно загрузить с вебсайта Wrox (*http://www.wrox.com*):

```
# Makefile для программ-примеров из
# "PostgreSQL. Основы"

# Редактирование основных каталогов для
# установки PostgreSQL

INC=/usr/local/pgsql/include
LIB=/usr/local/pgsql/lib

CFLAGS=-I$(INC)
LDLIBS=-L$(LIB) -lpq

ALL: async1 connect create cursor cursor2 print select1 select2          all:
$(ALL)

            clean  :
                    @rm -f  *.o *~  $(ALL)
```

Теперь можно создавать программу так:

```
$ make program
```

Дополнительная информация

Обратите внимание, что `PQstatus` и `PQfinish` могут принимать указатель `NULL` для дескриптора соединения, поэтому в данном случае мы не проверяем правильность результата, возвращаемого `PQconnectdb`.

Можно получить удобочитаемую строку, описывающую состояние соединения или произошедшую ошибку, вызвав `PQerrorMessage`:

```
char *PQerrorMessage(const PGconn *conn);
```

Функция возвращает указатель на строку описания. Эта строка будет перезаписана другими функциями `libpq`, поэтому ее необходимо прочитать или скопировать непосредственно после вызова `PqerrorMessage`, до вызова каких бы то ни было других функций `libpq`. Например, можно сделать сообщение о неудачной попытке соединения более информативным:

```
printf("connection failed: %s", PQerrorMessage(myconnection));
```

Для получения дополнительной информации о соединении после того, как оно было установлено, кто-то может попробовать обратиться непосредственно к элементам структуры `PGconn` (определенным в `libpq-fe.h`), но этого делать не стоит. Наш код, вероятно, перестанет работать в какой-нибудь из следующих версий `libpq`, если изменится внутренняя структура `PGconn`.

Сведения о соединении на самом деле могут быть очень важны, поэтому libpq предоставляет несколько функций доступа, возвращающих значения атрибутов соединения:

```
char *PQdb(const PGconn *conn);
char *PQuser(const PGconn *conn);
char *PQpass(const PGconn *conn);
char *PQhost(const PGconn *conn);
char *PQport(const PGconn *conn);
char *PQoptions(const PGconn *conn);
```

Эти функции возвращают соответственно имя базы данных, имя и пароль пользователя, имя сервера, номер порта сервера и параметры, относящиеся к соединению. Все эти значения не меняются в течение соединения.

Если с соединением возникают проблемы, можно попытаться переустановить его. Этим занимается функция PQreset. Она закроет соединение с сервером базы данных и попробует осуществить новое, с параметрами, действовавшими при задании первоначального соединения:

```
void PQreset(PGconn *conn);
```

Выполнение операторов SQL с помощью libpq

Следующим шагом после соединения с базой данных из программы, написанной на C, является выполнение операторов SQL. В действительности это достаточно просто. Начнем с функции PQexec:

```
PGresult *PQexec(PGconn *conn, const char *sql_string);
```

По существу, происходит следующее: мы передаем SQL-оператор функции PQexec, а сервер, к которому мы подключены посредством не-NULL-соединения conn, выполняет его. Результат сообщается через специальную структуру – PGresult. В редких случаях PQexec может вернуть указатель NULL, это будет означать, что не хватает памяти для новой структуры, передающей результаты. Даже если нет данных, которые должны быть возвращены, PQexec вернет действительный ненулевой указатель на результирующую структуру, не содержащую записей с данными.

Можно определить статус завершения оператора SQL, проанализировав результат функцией PQresultStatus, которая возвращает одно из значений, составляющих перечислимый тип ExecStatusType:

```
ExecStatusType PQresultStatus(const PGresult *result);
```

Типы статусов представлены в табл. 13.2:

Таблица 13.2. Статусы выполнения оператора SQL

Тип статуса	Описание
PGRES_EMPTY_QUERY	Доступ к базе данных не требуется. Обычно является результатом пустой строки запроса.
PGRES_COMMAND_OK	Успешное завершение. Команда не возвращает данные.
PGRES_TUPLES_OK	Успешное завершение. Запрос вернул ноль или более строк.
PGRES_BAD_RESPONSE	Ошибка, ответ сервера непонятен.
PGRES_NONFATAL_ERROR	Нефатальная ошибка, можно попробовать еще раз.
PGRES_FATAL_ERROR	Фатальная ошибка, повторить попытку нельзя.

Другие значения статуса указывают на некоторые непредвиденные проблемы с сервером, такие как его нахождение в процессе резервного копирования или в процессе останова.

Статус PGRES_EMPTY_QUERY часто указывает на проблему клиентской программы, посылающей запрос, в котором сервер просят ничего не делать.

Статус PGRES_COMMAND_OK означает, что оператор выполнен корректно и что он принадлежал к типу операторов, не возвращающих данные (например CREATE TABLE.)

Статус PGRES_TUPLES_OK указывает на то, что оператор выполнен и что он принадлежит к типу, возвращающему данные (например SELECT). Это не означает, что в данном случае данные возвращены. Необходимы дальнейшие запросы, чтобы определить, сколько данных на самом деле доступно.

Оставшиеся статусы: PGRES_BAD_RESPONSE, PGRES_NONFATAL_ERROR и PGRES_FATAL_ERROR означают, что оператор не был выполнен.

Приведем пример фрагмента программы, в котором для определения точных результатов вызова PQexec используется PQresultStatus:

```
PGresult *result;
result  = PQexec(myconnection,
"SELECT customer_id FROM customer");
switch(PQresultStatus(result)) {
    case PGRES_TUPLES_OK:
        /* выяснить, есть ли данные для обработки */
        if(PQntuples(result)) {
            /* обработать данные */
            break;
        }
        /* нет данных, перейти к случаю отсутствия данных */
    case PGRES_COMMAND_OK:
        /* все OK, нет данных для обработки */
        break;
```

```
        case PGRES_EMPTY_QUERY:
            /* серверу нечего делать, вероятно, ошибка? */
            break;
        case PGRES_NONFATAL_ERROR:
            /* можно продолжать, можно попытаться еще раз */
            break;
        case PGRES_BAD_RESPONSE:
    case PGRES_FATAL_ERROR:
    default:
            /* фатальная или неизвестная ошибка, нельзя продолжить */
    }
```

Более подробно о `PQntuples` будет рассказано чуть ниже в этой главе, когда речь пойдет о статусе `PGRES_TUPLES_OK` для оператора `SELECT`.

В поиске ошибок может помочь одна полезная функция под названием `PQresStatus`. Она преобразует код статуса результата в строку:

```
const char *PQresStatus(ExecStatusType status);
```

В случае ошибки можно получить более полное текстовое сообщение, вызвав функцию `PQresultErrorMessage`, подобно тому как мы это делали для соединений:

```
const char *PQresultErrorMessage(const PGresult *result);
```

Результирующие множества, как и соединения, должны быть освобождены после окончания работы с ними. Для этой цели походит функция `PQclear`, также обрабатывающая указатели `NULL`. Обратите внимание, что результаты не очищаются автоматически, даже при закрытии соединения, и при необходимости они могут храниться бесконечно:

```
void PQclear(PGresult *result);
```

Посмотрим на некоторые простые примеры выполнения операторов SQL. Создадим маленькую простую таблицу базы данных `test` для проведения испытаний. Позже произведем несколько операций с учебной таблицей `customer`, чтобы обеспечить возврат больших объемов данных.

Создадим в базе данных таблицу с именем `number`. Будем хранить в ней числа и их описание на английском языке. Таблица будет выглядеть так:

```
 value |    name
-------+-------------
    42 | The Answer
    29 | My Age
    66 | Clickety-Click
```

Чтобы создать таблицу и вставить в нее значения, надо просто вызвать функцию `PQexec` с соответствующей строкой, содержащей SQL-запрос,

который требуется выполнить. Программа будет содержать подобные вызовы функции:

```
PGconn *myconnection;
...
PQexec(myconnection,"CREATE TABLE number ( value INTEGER, name VARCHAR)");
PQexec(myconnection,"INSERT INTO number VALUES (42, 'The Answer')");
```

Следует позаботиться о возможных ошибках, например, если таблица уже существует, то при попытке создать ее будет получено сообщение об ошибке. Если при создании таблицы `number` оказывается, что она уже существует, то `PQresultErrorMessage` вернет строку такого содержания:

```
ERROR:  Relation 'number' already exists
```

Чтобы несколько упростить ситуацию, напишем собственную функцию, которая будет выполнять операторы SQL, проверять результаты и выводить ошибки. По мере продвижения вперед будем добавлять в нее новые возможности. Пока же займемся первой версией.

Теперь выполнение SQL-запросов будет для вас почти таким же простым, как ввод команд в `psql`. Сохраните следующий код в файле `create.c`:

```
#include<stdlib.h>
#include<libpq-fe.h>

void doSQL(PGconn *conn, char *command)
{
  PGresult *result;

  printf("%s\n", command);

  result = PQexec(conn, command);
  printf("status is %s\n", PQresStatus(PQresultStatus(result)));
  printf("result message: %s\n", PQresultErrorMessage(result));
  PQclear(result);
}

int main()
{
  PGresult *result;
  PGconn *conn;

  conn = PQconnectdb("");

  if(PQstatus(conn) == CONNECTION_OK) {
    printf("connection made\n");

/* doSQL(conn, "DROP TABLE number"); */
```

```
    doSQL(conn, "CREATE TABLE number (
                   value INTEGER,
                   name  VARCHAR
                 )");
    doSQL(conn, "INSERT INTO number values(42, 'The Answer')");
    doSQL(conn, "INSERT INTO number values(29, 'My Age')");
    doSQL(conn, "INSERT INTO number values(29, 'Anniversary')");
    doSQL(conn, "INSERT INTO number values(66, 'Clickety-Click')");
  }
  else
    printf("connection failed %s\n", PQerrorMessage(conn));

  PQfinish(conn);
  return EXIT_SUCCESS;
}
```

Создана таблица `number` и в нее добавлено несколько записей. Если еще раз запустить программу, будет выдано сообщение о фатальной ошибке, поскольку второй раз создать таблицу невозможно. Раскомментируйте команду `DROP TABLE` — теперь программа будет уничтожать и пересоздавать таблицу при каждом запуске.

Конечно, в рабочем коде такое бесцеремонное обращение с ошибками невозможно. Здесь же был пропущен возврат результата из `doSQL` (чтобы не загромождать программу), и мы могли двигаться вперед, не задумываясь о возможных неудачах.

Откомпилировав и запустив программу, вы должны увидеть выполняющуюся команду и некоторую информацию о статусе:

```
$ make create
$ PGDATABASE=bpsimple ./create
connection made
...
INSERT INTO number VALUES(66, 'Clickety-Click')
status is PGRES_COMMAND_OK
result message:
$
```

Для включения в оператор SQL указанных пользователем данных необходимо создать строку, содержащую требуемые значения, которая будет передана функции `PQexec`. Чтобы добавить все одноразрядные целые числа, можно написать:

```
for(n = 0; n < 10; n++) {
    sprintf(buffer,
"INSERT INTO number VALUES(%d, 'single digit')", n);
    PQexec(buffer);
}
```

Для обновления или удаления строк таблицы можно использовать команды UPDATE и DELETE:

```
UPDATE number SET name = 'Zaphod' WHERE value = 42
DELETE FROM number WHERE value = 29
```

Добавим в нашу программу соответствующие вызовы PQexec (или doSQL): сначала изменим текст описания для числа 42 (на Zaphod), а затем удалим обе записи для значения 29. Результат изменений можно проверить при помощи psql:

```
$ psql -d bpsimple
bpsimple=# SELECT * FROM number;
 value |       name
-------+-----------------
    66 | Clickety-Click
    42 | Zaphod

bpsimple=#
```

DELETE и UPDATE могут изменить более чем одну строку таблицы (или **кортеж**, PostgreSQL предпочитает такое название), поэтому часто бывает нелишним узнать, сколько строк было изменено. Эту информацию можно получить, вызвав функцию PQcmdTuples:

```
const char *PQcmdTuples(const PGresult *result);
```

Кого-то может удивить, что PQcmdTuples возвращает не целое число (как можно было ожидать), а строку, содержащую цифры. Очень просто модифицировать функцию doSQL так, чтобы она сообщала об измененных строках:

```
printf("#rows affected %s\n", PQcmdTuples(result));
```

Функция PQcmdTuples возвращает пустую строку для команд, которые не оказывают никакого влияния на строки (как CREATE TABLE), и строки "1" и "2" для тех, которые изменяют строки (команды INSERT и DELETE).

Необходимо быть очень внимательными, чтобы отличить команды, не воздействующие на строки, от тех, которым не удалось изменить их из-за ошибки выполнения. Обязательно следует проверять статус результата, чтобы распознать ошибки, а не только измененные строки.

Транзакции

Иногда требуется обеспечить выполнение группы команд SQL именно как группы, другими словами, чтобы либо все они произвели изменения в базе данных, либо ни одна из них этого не сделала (если в каком-то месте произойдет ошибка).

Как и в стандартном SQL, можно сделать это при помощи libpq, используя поддержку транзакций. Просто вызываем PQexec с операторами SQL, содержащими BEGIN, COMMIT и ROLLBACK:

```
PQexec(conn, "BEGIN WORK");

/* make changes */

if(we changed our minds) {
    PQexec(conn, "ROLLBACK WORK");
}
else {
    PQexec(conn, "COMMIT WORK");
}
```

Транзакции и блокировки подробно обсуждались в главе 9. Все описанные там возможности доступны и в программах libpq, при условии передачи соответствующей строки SQL-запроса функции PQexec.

Извлечение данных из запросов

До сих пор речь шла в основном об операторах SQL, которые не возвращают никаких данных. Теперь подумаем о том, как распорядиться данными, возвращаемыми функцией PQexec и представляющими собой результаты выполнения операторов SELECT.

Если SELECT выполняется посредством PQexec, то результирующее множество будет содержать информацию о данных, возвращенных запросом.

Обработка результатов запросов может показаться достаточно утомительной, т. к. не всегда точно известно, чего можно ожидать. При выполнении оператора SELECT заранее неизвестно, будут ли возвращены ноль строк, одна строка или же несколько миллионов строк. Если в SELECT используется шаблон (*), то неизвестно даже, какие столбцы и с каким названиями будут возвращены.

Как правило, программируя приложение, мы хотим, чтобы оно выбирало только определенные столбцы. Тогда, если схема базы данных изменится, – например, будут добавлены новые столбцы, то функция, не зависящая от нового столбца, будет продолжать работать, как и предполагалось.

Иногда же (например, если мы пишем универсальную программу на SQL, которая будет получать операторы от пользователя и выводить результаты) бывает удобнее обратиться к более общему решению, и с помощью libpq это возможно. Надо лишь познакомиться с еще несколькими функциями.

Когда PQexec успешно выполняет оператор SELECT, мы ожидаем увидеть в качестве статуса результата PGRES_TUPLES_OK. Следующим шагом долж-

но стать определение количества строк, входящих в результирующее множество. Для этого вызовем функцию PQntuples:

```
int PQntuples(const PGresult *result);
```

Будет выведено общее количество строк результата, которое, конечно же, может равняться и нулю.

Количество полей (атрибутов или столбцов) кортежа можно узнать, вызвав функцию PQnfields:

```
int PQnfields(const PGresult *result);
```

Поля результата пронумерованы начиная с нуля, можно извлечь и их имена, используя PQfname:

```
char *PQfname(const PGresult *result, int index);
```

Размер поля выдается функцией PQfsize:

```
int PQfsize(const PGresult *result, int index);
```

Для полей фиксированной длины PQfsize возвращает количество байт, занимаемых значением в этом конкретном столбце. Для полей переменной длины PQfsize вернет -1.

Если вдруг понадобится, можно узнать порядковый номер столбца с заданным именем, вызвав PQfnumber:

```
int PQfnumber(const PGresult *result, const char *field);
```

Изменим нашу функцию doSQL так, чтобы она выводила некоторую информацию о данных, возвращенных SELECT-запросом. Вот ее вторая версия:

```
void doSQL(PGconn *conn, char *command);
{
  PGresult *result;

  printf("%s\n", command);

  result = PQexec(conn, command);
  printf("status is %s\n", PQresStatus(PQresultStatus(result)));
  printf("#rows affected %s\n", PQcmdTuples(result));
  printf("result message: %s\n", PQresultErrorMessage(result));

  switch(PQresultStatus(result)) {
  case PGRES_TUPLES_OK:
    {
      int n = 0;
      int nrows = PQntuples(result);
      int nfields = PQnfields(result);
      printf("number of rows returned = %d\n", nrows);
```

```
        printf("number of fields returned = %d\n", nfields);
        /* Print the field names */
        for(n = 0; n < nfields; n++) {
            printf(" %s:%d",
                   PQfname(result, n), PQfsize(result, n));
        }
        printf("\n");
      }
    }
    PQclear(result);
}
```

Теперь при выполнении оператора SELECT можно видеть свойства возвращаемых данных:

```
doSQL(conn, "SELECT * FROM number WHERE value = 29");
```

Результатом такого вызова будут следующие выходные данные:

```
status is PGRES_TUPLES_OK
#rows affected
result message:
number of rows returned = 2
number of fields returned = 2
 value:4 name:-1
```

Заметьте, что пустая строка возвращается функцией PQcmdTuples для запросов, которые не могут изменить строки, а PQresultErrorMessage возвращает пустую строку, если не было ошибок. Теперь мы готовы извлечь данные из полей, возвращенных в строки результирующего множества. Строки пронумерованы с нуля.

Обычно все данные передаются с сервера в виде символьных строк. Символьное представление данных можно получить, вызвав функцию PQgetvalue:

```
char *PQgetvalue(const PGresult *result, int tuple, int field);
```

Чтобы заранее узнать длину строки, возвращаемой PQgetvalue, можно вызвать PQgetlength:

```
int PQgetlength(const PGresult *result, int tuple, int field);
```

Как уже говорилось, и кортежи (строки) и поля (столбцы) нумеруются с нуля.

Добавим в функцию doSQL вывод некоторых данных:

```
void doSQL(PGconn *conn, char *command)
{
  PGresult *result;

  printf("%s\n", command);
```

```
    result = PQexec(conn, command);
    printf("status is %s\n", PQresStatus(PQresultStatus(result)));
    printf("#rows affected %s\n", PQcmdTuples(result));
    printf("result message: %s\n", PQresultErrorMessage(result));

    switch(PQresultStatus(result)) {
    case PGRES_TUPLES_OK:
      {
        int r, n;
        int nrows = PQntuples(result);
        int nfields = PQnfields(result);
        printf("number of rows returned = %d\n", nrows);
        printf("number of fields returned = %d\n", nfields);
        for(r = 0; r < nrows; r++) {
        for(n = 0; n < nfields; n++)
          printf(" %s = %s(%d),",
           PQfname(result, n),
           PQgetvalue(result, r, n),
           PQgetlength(result, r, n));
        printf("\n");
        }
      }
    }
    PQclear(result);
  }
```

Выводится полный результат запроса `SELECT`, в том числе длины строк, содержащих данные:

```
SELECT * FROM number WHERE value = 29
Status is PGRES_TUPLES_OK
#rows affected
result message:
number of rows returned = 2
number of fields returned = 2
 value = 29(2), name = My Age(6),
 value = 29(2), name = Anniversary(11),
```

Обратите внимание, что длина строки данных не включает в себя завершающий ноль, который присутствует в строке, возвращенной `PQgetvalue`.

Строковые данные, аналогичные используемым в столбцах, определенных как `CHAR(n)`***, дополняются пробелами. Это может привести к неожиданным результатам при поиске некоторого строкового значения или сравнении значений для сортировки. Если значение «Zaphod» вставляется в столбец, определенный как*** `CHAR(8)`***, то обратно будет получено*** `"Zaphod<пробел><пробел>"`***, то есть строка, которая при сравнении не будет признана тождественной строке*** `"Zaphod"` ***(если используется библиотечная функция C*** `strcmp`***). Известно, что эта небольшая «неувязочка» оказывалась ловушкой для некоторых даже очень опытных разработчиков.***

Прежде чем двигаться дальше, надо обсудить одно небольшое затруднение. Тот факт, что результаты запроса возвращаются в виде символьных строк, означает, что невозможно сразу различить пустую строку и NULL-значение SQL.

К счастью, библиотека libpq предоставляет функцию, вызвав которую, можно определить, является ли значение поля в результирующем кортеже равным NULL:

```
int PQgetisnull(const PGresult *result, int tuple, int field);
```

Следует вызывать PQgetisnull, если извлекается поле, значение которого может быть неизвестной величиной (NULL). Она возвращает 1, если поле содержит NULL, и 0 – в противном случае. Тогда внутренний цикл программы из последнего примера будет выглядеть так:

```
        for(n = 0; n < nfields; n++) {
          if(PQgetisnull(result, r, n))
              printf(" %s is NULL,", PQfname(result, n));
           else
            printf(" %s = %s(%d),",
             PQfname(result, n),
             PQgetvalue(result, r, n),
             PQgetlength(result, r, n));
          }
```

Вывод результатов запроса

Мы рассмотрели функции, обеспечивающие выполнение запросов и извлечение данных из базы данных PostgreSQL. Если требуется просто вывести результаты, можно обратиться к функции печати, предоставляемой libpq, которая выводит результирующие множества в довольно простом виде.

Функция PQprint форматирует результат в табличную форму, подобную используемой psql, и посылает его в указанный выходной поток. Функция выглядит следующим образом:

```
void PQprint(FILE *output, const PGresult *result, const PQprintOpt *options);
```

Однако PQprint больше не поддерживается разработчиками PostgreSQL, поэтому не следует полагаться на нее в коммерческих разработках. Для тех, кто захочет опробовать функцию: ее аргументами являются дескриптор открытого файла (output) для вывода, результирующее множество (result) и указатель на структуру, которая содержит параметры, задающие формат вывода (options). Подробнее о структуре:

```
struct {
    pqbool header;    /* выводить в качестве заголовка названия столбцов */
    pqbool align;     /* дополнять значения пробелами для выравнивания */
```

```
    pqbool html3;       /* форматировать как таблицу HTML */
    pqbool expanded;    /* развернуть таблицы */
    pqbool pager;       /* при необходимости использовать для вывода
                           постранично */
    char *fieldSep;     /* разделитель полей */
    char *tableOpt;     /* параметры для таблицы HTML - место в <TABLE …> */
    char *caption;      /* HTML <заголовок> */
    char **fieldName;   /* Замещающее множество имен полей */
} PQprintOpt;
```

Элементы структуры `PQprintOpt` весьма просты и понятны. Элемент `header`, будучи установлен в ненулевое значение, указывает, что первая строка таблицы вывода должн состоять из названий полей, которые могут быть изменены при помощи установки списка строк `fieldName`.

Каждая строка выходной таблицы состоит из значений полей, разделенных строкой `fieldSep` и дополненных пробелами для выравнивания (по отношению к другим строкам), если `align` не установлен в 0.

Вывод может выглядеть так:

```
+-------------+-------+-----------+--
| customer_id | title | fname     | town         |  zipcode |    phone |
+-------------+-------+-----------+--
|           1 | Miss  | Jenny     | Hightown     | NT2 1AQ  | 023 9876 |
+-------------+-------+-----------+--
|           3 | Miss  | Alex      | Nicetown     | NT2 2TX  | 010 4567 |
+-------------+-------+-----------+--
```

Если ожидается очень длинный вывод, то можно задать ненулевое значение для `pager`, чтобы разбить выходные данные на страницы, тогда они будут передаваться через фильтр, который приостановит вывод, например после каждой страницы. Если ненулевое значение указано для `expanded`, то формат вывода изменяется следующим образом: каждое поле каждой строки выводится на отдельной строке.

Можно задать HTML-вывод, подходящий для включения в веб-страницу, установив `html3` в ненулевое значение. Можно указать параметры и заголовок таблицы, задав строки `tableOpt` и `caption`.

Приведем в качестве примера программу (`print.c`), которая использует `PQprint` для генерации HTML-вывода:

```
#include <stdlib.h>
#include <libpq-fe.h>

int main()
{
  PGresult *result;
  PGconn *conn;

  conn = PQconnectdb("");
```

```
  if(PQstatus(conn) == CONNECTION_OK) {
    printf("connection made\n");

    result = PQexec(conn, "SELECT * FROM customer
                           WHERE town = 'Bingham'");
    {
      PQprintOpt pqp;
      pqp.header = 1;
      pqp.align = 1;
      pqp.html3 = 1;
      pqp.expanded = 0;
      pqp.pager = 0;
      pqp.fieldSep = "";
      pqp.tableOpt = "align=center";
      pqp.caption = "Bingham Customer List";
      pqp.fieldName = NULL;
      printf("<HTML><HEAD><TITLE>Customers</TITLE></HEAD><BODY>\n");
      PQprint(stdout, result, &pqp);
      printf("</BODY></HTML>\n");
    }

  }

  PQfinish(conn);
  return EXIT_SUCCESS;
}
```

Выходными данными программы, представленной выше, является HTML-код, который отображается на экране (`stdout`). Вот что мы увидим:

```
$ PGDATABASE=bpsimple  ./print

<HTML><HEAD></HEAD><BODY>
<table align=center><caption align=high>Bingham Customer List</caption>
<tr><th align=right>customer_id</th><th align=left>title</th><th
align=left>fnam
e</th><th align=left>lname</th><th align=left>addressline</th><th
align=left>tow
n</th><th align=left>zipcode</th><th align=right>phone</th></tr>
<tr><td align=right>7</td><td align=left>Mr  </td><td align=left>Richard</
td><td
 align=left>Stones</td><td align=left>34 Holly Way</td><td
align=left>Bingham</t
d><td align=left>BG4 2WE   </td><td align=right>342 5982</td></tr>
<tr><td align=right>8</td><td align=left>Mrs </td><td align=left>Ann</td><td
ali
gn=left>Stones</td><td align=left>34 Holly Way</td><td align=left>Bingham</
td><t
d align=left>BG4 2WE   </td><td align=right>342 5982</td></tr>
```

```
<tr><td align=right>11</td><td align=left>Mr  </td><td align=left>Dave</
td><td a
lign=left>Jones</td><td align=left>54 Vale Rise</td><td align=left>Bingham</
td><
td align=left>BG3 8GD   </td><td align=right>342 8264</td></tr>
</table>
</BODY></HTML>
```

Чтобы иметь возможность просмотреть это в броузере, надо просто перенаправить вывод программы в файл (например `list.html`), а затем просмотреть этот файл. Перенаправление вывода выполняется следующей командой:

```
$ PGDATABASE=bpsimple ./print > list.html
```

Вот как выглядит HTML-страница в броузере (рис. 13.1):

Рис. 13.1. Просмотр в броузере файла, в который перенаправлен сгенерированный HTML-вывод

Курсоры

При написании сложных приложений для реальной работы иногда приходится работать с очень большими объемами данных. База данных PostgreSQL может хранить таблицы, содержащие огромное количество строк.

Однако когда дело доходит до обработки результатов запроса, породившего большой объем данных, ее «отдают на откуп» клиентскому приложению и его рабочей среде. Настольный компьютер вполне может испытывать затруднения, имея дело с миллионами кортежей, возвращенными одновременно в качестве результата выполнения одного оператора `SELECT`. Большое результирующее множество в состоянии поглотить значительную часть памяти, а если работа происходит по сети, то и значительную часть пропускной способности, и его передача занимает вполне ощутимое время.

На самом деле нам надо выполнять запрос и обрабатывать его результаты по частям. Например, если в приложении требуется вывести полный список клиентов, то можно извлечь все эти данные за один шаг. Но разумнее выбирать их страницами, скажем, по 25 записей на каждой, и выводить в приложении страница за страницей.

В libpq это можно сделать, применив курсоры. Курсоры – это универсальный способ обработки неизвестного количества возвращаемых строк. При поиске почтового индекса, особенно заданного пользователем, заранее неизвестно, сколько записей будет возвращено: ноль, одна или множество.

Вообще, следует избегать написания программы, которая предполагает, что оператор SELECT вернет или одну строку, или же ни одной, если только этот оператор не является простой агрегатной функцией, такой как запрос SELECT COUNT(*) FROM, или если SELECT выполняется для первичного ключа, когда можно быть уверенным в том, что результатом всегда будет ровно одна строка. Если же есть сомнения, используйте курсор.

Чтобы разобраться с множеством строк, возвращаемых запросом, будем извлекать их по одной (или по несколько) за раз посредством FETCH, при этом значения столбцов будут попадать в результирующее множество тем же способом, что и для операторов SELECT, результаты которых извлекаются все одновременно.

Объявим курсор, который будет применяться для прохода по набору возвращенных строк. Курсор будет работать как закладка, а выборка строк не прекратится до тех пор, пока не закончатся данные.

Чтобы применить курсор, надо объявить его и указать запрос, с которым он связан. Объявить курсор можно только внутри транзакции PostgreSQL, поэтому также необходимо начать транзакцию:

```
PQexec(conn, "BEGIN work");
PQexec(conn, "DECLARE mycursor CURSOR FOR SELECT ..."
```

Теперь можно начинать извлечение строк результата. Делаем это, выполняя FETCH для одновременной выборки некоторого количества строк (в том числе и всех оставшихся):

```
result = PQexec(conn, "FETCH 1 IN mycursor");
result = PQexec(conn, "FETCH 4 IN mycursor");
result = PQexec(conn, "FETCH ALL IN mycursor");
```

Когда все строки будут выбраны, результирующее множество сообщит о том, что строк больше нет.

Закончив работу с курсором, закрываем его и завершаем транзакцию:

```
PQexec(conn, "COMMIT work");
PQexec(conn, "CLOSE mycursor");
```

Рассмотрим в качестве примера программу, которая запрашивает и обрабатывает список клиентов из базы данных bpsimple, делая это постранично с применением курсора.

Общий подход заключается в том, чтобы организовать программу следующим образом:

```
#include <libpq-fe.h>

main()
{
    /* Соединение с базой данных PostgreSQL */

    /* Создание курсора для оператора SQL SELECT */
    DO
            /* Выборка партий результатов запроса */
            /* Обработка результатов запроса */

        UNTIL no more results
        /* Закрытие курсора */

    /* Отключение от базы */
}
```

Для каждой из выбираемых партий результатов запроса мы получаем доступ к указателю PGresult, с которым можно работать так же, как и раньше. Давайте посмотрим на курсор в действии (в cursor.c), сначала получив все результаты сразу, чтобы проверить правильность операторов SQL:

```
#include <stdlib.h>
#include <libpq-fe.h>

void doSQL(PGconn *conn, char *command)
{
  PGresult *result;
printf("%s\n", command);

  result = PQexec(conn, command);
  printf("status is %s\n", PQresStatus(PQresultStatus(result)));
  printf("#rows affected %s\n", PQcmdTuples(result));
  printf("result message: %s\n", PQresultErrorMessage(result));

  switch(PQresultStatus(result)) {
  case PGRES_TUPLES_OK:
    {
      int r, n;
      int nrows = PQntuples(result);
      int nfields = PQnfields(result);
      printf("number of rows returned = %d\n", nrows);
      printf("number of fields returned = %d\n", nfields);
      for(r = 0; r < nrows; r++) {
```

```
    for(n = 0; n < nfields; n++)
      printf(" %s = %s(%d),",
         PQfname(result, n),
         PQgetvalue(result, r, n),
         PQgetlength(result, r, n));
    printf("\n");
      }
    }
  }
  PQclear(result);
}

int main()
{
  PGresult *result;
  PGconn *conn;

  conn = PQconnectdb("");

  if(PQstatus(conn) == CONNECTION_OK) {
    printf("connection made\n");

    doSQL(conn, "BEGIN work");
    doSQL(conn, "DECLARE mycursor CURSOR FOR "
                          "SELECT fname, lname FROM customer");
    doSQL(conn, "FETCH ALL IN mycursor");
    doSQL(conn, "COMMIT work");
    doSQL(conn, "CLOSE mycursor");

  }
  else
    printf("connection failed\n");

  PQfinish(conn);
  return EXIT_SUCCESS;
}
```

Выполнив эту программу, вы увидите, как сразу будут выведены все клиенты:

```
connection made
DECLARE mycursor CURSOR FOR SELECT fname, lname FROM customer
status is PGRES_COMMAND_OK
#rows affected
result message:
BEGIN work
status is PGRES_COMMAND_OK
#rows affected
result message:
```

```
FETCH ALL IN mycursor
status is PGRES_TUPLES_OK
#rows affected
result message:
number of rows returned = 15
number of fields returned = 2
 fname = Jenny(5), lname = Stones(6),
 fname = Andrew(6), lname = Stones(6),
 fname = Adrian(6), lname = Matthew(7),
 fname = Simon(5), lname = Cozens(6),
 fname = Neil(4), lname = Matthew(7),
 fname = Richard(7), lname = Stones(6),
 fname = Ann(3), lname = Stones(6),
 fname = Christine(9), lname = Hickman(7),
 fname = Mike(4), lname = Howard(6),
 fname = Dave(4), lname = Jones(5),
 fname = Richard(7), lname = Neill(5),
 fname = Laura(5), lname = Hendy(5),
 fname = Bill(4), lname = O'Neill(7),
 fname = David(5), lname = Hudson(6),
 fname = Alex(4), lname = Matthew(7),
COMMIT work
status is PGRES_COMMAND_OK
#rows affected
result message:
CLOSE mycursor
status is PGRES_COMMAND_OK
#rows affected
result message:
```

Изменим программу так, чтобы иметь дело с результатами, скажем, по четыре за раз. Для этого необходимо иметь возможность сообщить о том, что все результаты извлечены. Когда обрабатывались все результаты сразу, сделать это было просто, т. к. `PQntuples` **сообщала, сколько результатов входит в множество. Если получать результаты партиями по четыре, тогда** `PQntuples` **будет возвращать 4 для каждой партии, кроме последней, в которую будет входить менее четырех элементов, а может быть, и ноль. Используем эту проверку (в** `cursor2.c`**). Поскольку функция** `doSQL` **не возвращает результатов, будем оперировать партиями непосредственно с помощью** `PQexec`**:**

```
#include <stdlib.h>
#include <libpq-fe.h>
void printTuples(PGresult *result)
{
      int r, n;
      int nrows = PQntuples(result);
      int nfields = PQnfields(result);
      printf("number of rows returned = %d\n", nrows);
printf("number of fields returned = %d\n", nfields);
```

```
      for(r = 0; r < nrows; r++) {
    for(n = 0; n < nfields; n++)
      printf(" %s = %s(%d),",
         PQfname(result, n),
         PQgetvalue(result, r, n),
         PQgetlength(result, r, n));
    printf("\n");
      }
}

void doSQL(PGconn *conn, char *command)
{
  PGresult *result;

  printf("%s\n", command);

  result = PQexec(conn, command);
  printf("status is %s\n", PQresStatus(PQresultStatus(result)));
  printf("#rows affected %s\n", PQcmdTuples(result));
  printf("result message: %s\n", PQresultErrorMessage(result));

  switch(PQresultStatus(result)) {
  case PGRES_TUPLES_OK:
    printTuples(result);
    break;
  }
  PQclear(result);
}

int main()
{
  PGresult *result;
  PGconn *conn;
  int ntuples = 0;

  conn = PQconnectdb("");

  if(PQstatus(conn) == CONNECTION_OK) {
    printf("connection made\n");

    doSQL(conn, "BEGIN work");
    doSQL(conn, "DECLARE mycursor CURSOR FOR "
                        "SELECT fname, lname FROM customer");
    do {
      result = PQexec(conn, "FETCH 4 IN mycursor");
      if(PQresultStatus(result) == PGRES_TUPLES_OK) {
        ntuples = PQntuples(result);
        printTuples(result);
```

```
        PQclear(result);
      }
      else ntuples = 0;
    } while(ntuples);

    doSQL(conn, "CLOSE mycursor");
    doSQL(conn, "COMMIT work");

  }
  else
    printf("connection failed\n");

  PQfinish(conn);
  return EXIT_SUCCESS;
}
```

В данной версии программы вы можете видеть несколько фрагментов результата, состоящих из четырех элементов, которые обрабатывались одновременно, и один неполный фрагмент:

```
connection made
...
DECLARE mycursor CURSOR FOR SELECT fname, lname FROM customer
status is PGRES_COMMAND_OK
#rows affected
result message:
number of rows returned = 4
number of fields returned = 2
 fname = Jenny(5), lname = Stones(6),
 fname = Andrew(6), lname = Stones(6),
 fname = Adrian(6), lname = Matthew(7),
 fname = Simon(5), lname = Cozens(6),
number of rows returned = 4
number of fields returned = 2
 fname = Neil(4), lname = Matthew(7),
 fname = Richard(7), lname = Stones(6),
 fname = Ann(3), lname = Stones(6),
 fname = Christine(9), lname = Hickman(7),
number of rows returned = 4
number of fields returned = 2
 fname = Mike(4), lname = Howard(6),
 fname = Dave(4), lname = Jones(5),
 fname = Richard(7), lname = Neill(5),
 fname = Laura(5), lname = Hendy(5),
number of rows returned = 3
number of fields returned = 2
 fname = Bill(4), lname = O'Neill(7),
 fname = David(5), lname = Hudson(6),
 fname = Alex(4), lname = Matthew(7),
number of rows returned = 0
```

```
number of fields returned = 2
COMMIT work
status is PGRES_COMMAND_OK
#rows affected
result message:
...
$
```

Можно использовать все параметры FETCH, описанные ранее, для получения результатов по одному (параметр NEXT), партиями (указав количество) или всех сразу (параметр ALL).

Двоичные значения

Все значения, с которыми мы имели дело в этой главе, передавались в виде символьных строк. Создавались операторы INSERT со строковыми значениями, и строковые же значения извлекались из результатов SELECT.

Можно подумать, что заниматься преобразованиями форматов слишком расточительно и лучше было бы иметь двоичный интерфейс. Если требуется вставить или выбрать значение с плавающей точкой, например для среднего потребления топлива, то не лучше ли передавать его прямо из и в тип C double?

Ответом будет: «И да, и нет». PostgreSQL поддерживает (в libpq) некоторые возможности работы с двоичными значениями, но полученная выгода будет весьма незначительной. В настоящее время можно извлекать двоичные значения только посредством курсора, в объявление которого включен параметр BINARY:

```
DECLARE mycursor BINARY CURSOR FOR...
```

Тогда функция PQbinaryTuples подтвердит, что результирующее множество, возвращенное с помощью FETCH данного курсора, содержит двоичные данные:

```
int PQbinaryTuples(const PGresult *result);
```

Если применяются двоичные кортежи, то PQgetvalue будет возвращать не строковый указатель, а указатель на двоичное представление поля, в двоичном формате, присущем серверу. При этом обработка данных переменной длины и порядка следования байтов многобайтовых значений (например денежных единиц) может вскоре вызвать затруднения. Рекомендуется придерживаться строкового представления при использовании libpq. В следующей главе в ходе обсуждении ecpg будет рассказано о том, как можно несколько упростить работу с двоичными значениями при помощи встроенного SQL.

Асинхронность

Во всех приведенных примерах функции libpq работали в режиме блокировки. Проще говоря, программы вызывали функцию libpq и ждали, пока она вернет значения. Мы уже знаем, что может быть сгенерирован большой объем данных, получение которого занимает некоторое время. В однопоточном (обычном) приложении, если применяется PQexec, то ничего нельзя делать до тех пор, пока не будет завершена PQexec. Пользователь должен ждать. Также сложно отменить запрос, если его результаты больше не требуются. Для большинства программ этого достаточно, и небольшое ожидание не создает проблем.

Вероятно, потребуется лишь выбирать результаты по несколько штук сразу с помощью курсора. В данном разделе представлена более сложная технология, позволяющая при необходимости получить полный контроль над поведением приложения.

Альтернативой режиму блокировки, который обычно устанавливается для приложений с графическим пользовательским интерфейсом, является режим без блокировки. В этом типе программ, когда что-то происходит, вы посылаете сообщения, и приложение должно на них отвечать. Так, в приложениях, ориентированных на работу с мышью, нам нужны сообщения о том, что пользователь нажал кнопку мыши или ввел данные в поле, или переместил окно и т. д.

Основой структуры программы без блокировок обычно бывает цикл обработки событий:

```
main()
{
    LOOP:
        /* ждать, пока произойдет событие */
        /* узнать, какое событие произошло */
        switch(event type) {
            /* обработать событие */
    }
}
```

Мы, по существу, ничего не делаем до тех пор, пока не произойдет какое-то событие. Получаем уведомление о каком-либо событии, узнаем, что именно произошло, и выполняем соответствующее действие.

Иногда язык программирования, на котором мы пишем, скрывает цикл обработки событий, и остается лишь сделать так, чтобы конкретная написанная нами функция вызывалась, когда происходит конкретное событие. Эти функции часто называют обратными вызовами, и главный цикл, являющийся основой приложения, осуществляет обратный вызов, когда ему есть что сказать. Так все устроено, например, в Visual Basic.

В libpq PostgreSQL обеспечивает некоторую поддержку такого способа программирования без блокировок. Его называют асинхронной работой, потому что операции базы данных, выполняющиеся на сервере, не синхронизируются с клиентским приложением, которое может выполнять другие задачи, а не ждать. Вместо ожидания следует спросить PostgreSQL, произошло ли что-нибудь, и в случае положительного ответа возобновить обработку.

Давайте посмотрим, как выполнить запрос в асинхронном режиме. Используем две функции, применяемые PQexec для выполнения ее работы, PQsendQuery и PQgetResult. Идея заключается в том, что PQsendQuery вызывается для того, чтобы отправить наш запрос на сервер. Как только запрос окажется в пути, можно начать заниматься другими вещами. Если сервер ответит незамедлительно, то результирующее множество будет сохранено в ожидании того момента, когда мы будем готовы извлечь его. Подготовившись, вызывайте функцию PQgetResult один или несколько раз, чтобы получать результаты запроса по мере их поступления:

```
int PQsendQuery(PGconn *conn, const char *query);
```

Функция PQsendQuery возвращает ноль, если запрос не отправлен, и устанавливает состояние ошибки, которое может быть получено при помощи функции PqerrorMessage. Иначе она возвращает 1, что означает успешную отсылку запроса:

```
PGresult *PQgetResult(PGconn *conn);
```

PQgetResult будет возвращать результирующее множество при каждом вызове до тех пор, пока все результаты активного запроса не будут возвращены. Результирующее множество может не содержать ни одного кортежа, если дополнительные данные пока еще недоступны, но это не означает, что все данные уже получены. Если же все данные действительно получены, то PQgetResult возвращает указатель NULL. После окончания работы с результатами запроса каждое результирующее множество должно быть очищено, для чего вызывается PQclear.

Необходимо убедиться, что ни PQsendQuery, ни PQgetResult не заблокируют приложение и не заставят ждать завершения их выполнения. Чтобы посодействовать PQsendQuery, можно установить само соединение в состояние без блокировок, вызвав PQsetnonblocking:

```
int PQsetnonblocking(PGconn *conn, int arg);
```

Чтобы предотвратить блокировку PQsendQuery, осуществляем вызов функции PQsetnonblocking с ненулевым аргументом. Тогда PQsendQuery вернет ошибку, если она будет заблокирована, и приложение сможет повторить попытку впоследствии.

PQsetnonblocking возвращает ноль в случае удачной работы и -1, если при изменении режима соединения возникли проблемы. Режим блокировки соединения можно проверить, вызвав PQisnonblocking:

```
int PQisnonblocking(const PGconn *conn);
```

Если применяется режим без блокировок, то будет возвращено ненулевое значение.

Для того чтобы выполнить такую операцию без блокировок, нужно запрограммировать что-то подобное (async1.c):

```
#include <stdlib.h>
#include <libpq-fe.h>

void printTuples(PGresult *result)
{
      int r, n;
      int nrows = PQntuples(result);
      int nfields = PQnfields(result);
      printf("number of rows returned = %d\n", nrows);
      printf("number of fields returned = %d\n", nfields);
      for(r = 0; r < nrows; r++) {
    for(n = 0; n < nfields; n++)
      printf(" %s = %s(%d),",
         PQfname(result, n),
         PQgetvalue(result, r, n),
         PQgetlength(result, r, n));
    printf("\n");
      }
}

int main()
{
  PGresult *result;
  PGconn *conn;

  conn = PQconnectdb("");

  if(PQstatus(conn) == CONNECTION_OK &&
     PQsetnonblocking(conn,1) == 0) {
    printf("connection made\n");

    PQsendQuery(conn, "SELECT * FROM customer");
    while(result = PQgetResult(conn)) {
      printTuples(result);
      PQclear(result);
    }
  }
```

```
    else
      printf("connection failed\n");

    PQfinish(conn);
    return EXIT_SUCCESS;
  }
```

Эта программа отсылает запрос, а затем собирает результаты, не применяя блокировки. Как и при использовании курсора (обсуждалось ранее), теперь нет простого способа сообщить, сколько всего строк будет возвращено, но на практике это редко вызывает затруднения.

Тем не менее изредка случается такое стечение обстоятельств, при котором PQgetResult может осуществить блокировку, например, если сервер базы данных занят. Эту ситуацию можно обойти при помощи низкоуровневых функций libpq. Если необходим полный контроль над соединением, а выполняемые запросы настолько сложны, что могут занять сервер на достаточно долгое время, обратите внимание на функции PQisBusy, PQconsumeInput и PQflush в руководстве программиста по PostgreSQL, включенном в дистрибутив с исходными текстами:

```
int PQisBusy(PGconn *conn);
```

PQisBusy возвращает 1, если текущий запрос занят, и PQgetResult была бы заблокирована, если бы ее вызвали:

```
int PQflush(PGconn *conn);
```

PQflush пытается отправить необработанные данные, ожидающие попадания на сервер. Она возвращает ноль, если удалось успешно обработать всю очередь запросов или она уже была пуста:

```
int PQconsumeInput(PGconn *conn);
```

PQconsumeInput — это функция libpq, которая передает данные, ожидающие считывания при соединении с базой данных, во внутренние структуры данных libpq. Она обычно вызывается такими функциями, как PQexec, но ее можно вызвать и явно, если необходим контроль над поведением приложения в том, что касается блокировок.

Если в приложении применяется системный вызов SELECT для реагирования на события чтения и записи в файловые дескрипторы или сетевые сокеты, то можно включить в SELECT и соединение с PostgreSQL.

Для того чтобы сделать это, необходимо получить сокет, используемый соединением с базой данных. Получим его из функции PQsocket:

```
int PQsocket(const PGconn *conn);
```

Сокет будет подавать сигнал активности в случае наличия на сервере данных для обработки, но для того чтобы он мог работать, надо обеспе-

чить выполнение нескольких условий. Во-первых, не должно быть данных, ожидающих отсылки на сервер. Для выполнения этого условия будем вызывать PQflush до тех пор, пока она не вернет ноль. Во-вторых, данные должны быть считаны из соединения функцией PQconsumeInput до того, как будет вызвана PQgetResult.

Если требуется отменить запрос до того, как прочитаны все его результаты, вызовем функцию PQcancelRequest – она отправит серверу указание остановить обработку запроса:

```
int PQcancelRequest(PGconn *conn);
```

PQcancelRequest вернет 1, если ей удалось отправить указание на отмену, и 0 – если не удалось. Вероятно, потому, что запрос уже был завершен. Отмененный запрос проявит себя в виде ошибки результирующего множества. Вне зависимости от результата выполнения PQcancelRequest можно опоздать с остановкой вывода результатов и не увидеть никаких ошибок в результирующем множестве.

Применение PQcancelRequest полезно при работе в режиме без блокировок, но ее можно вызывать и из обработчика сигналов для завершения долго выполняющегося запроса при соединении с блокировкой. Можно также установить для исходного соединения с базой данных режим без блокировок, применив PQconnectStart и PQconnectPoll:

```
PGconn *PQconnectStart(const char *conninfo);
PostgresPollingStatusType *PQconnectPoll(PGconn *conn);
```

PQconnectStart напоминает функцию PQconnectdb, с тем лишь отличием, что она возвращает результат сразу же, прежде чем установлено соединение, запрошенное в строке conninfo. До тех пор пока параметр имени сервера не будет разрешен с применением DNS, функция PQconnectStart не будет блокироваться.

Прежде чем использовать новое соединение, необходимо убедиться, что оно к этому готово. Для начала следует вызвать функцию PQstatus, проверяя, не потерпел ли неудачу вызов PQconnectStart, оставив соединение со статусом CONNECTION_BAD. Затем необходимо проверить состояние соединения с помощью PQconnectPoll, которая не осуществляет блокировку. Функция PQconnectPoll может возвращать следующие результаты:

```
PGRES_POLLING_FAILED    /*соединение не удалось установить*/
PGRES_POLLING_OK        /*соединение установлено*/
```

Не прекращая проверку, пока результат не равен PGRES_POLLING_FAILED, и до тех пор, пока он не станет равным PGRES_POLLING_OK, можно выявить окончание процесса установки соединения в режиме без блокировок.

Приведем пример программы (async2.c), реализующей асинхронное соединение с базой данных:

```
#include <stdlib.h>
#include <libpq-fe.h>

int main()
{
  PGresult *result;
  PGconn *conn;

  /* Начало асинхронного соединения */
  conn = PQconnectStart("");

  if(PQstatus(conn) == CONNECTION_BAD) {
    printf(" cannot start connect: %s\n", PQerrorMessage(conn));
  }
  else {
    /* выполнять какую-то работу, периодически вызывая PQconnectPoll */
    PostgresPollingStatusType status;
    do {
      printf("polling\n");
      status = PQconnectPoll(conn);
    }
    while(status != PGRES_POLLING_FAILED &&
      status != PGRES_POLLING_OK);

    if(status == PGRES_POLLING_OK)
      printf("connection made!\n");
    else
      printf("connection failed: %s\n", PQerrorMessage(conn));
  }
  PQfinish(conn);
  return EXIT_SUCCESS;
}
```

Запустив программу, мы увидим множество сообщений о проверке состояния, прежде чем появится уведомление об установлении или неустановлении соединения:

```
$ PGDATABASE=bpsimple ./async2
polling
polling
...
connection made!
$
```

Резюме

В данной главе обсуждалось создание приложений PostgreSQL на языке C. В следующих главах вы узнаете, какие интерфейсы для PostgreSQL имеются в других языках программирования.

Было рассказано о том, как библиотека `libpq` предоставляет доступ к низкоуровневым функциям PostgreSQL, позволяя подключиться к базе данных, находящейся на локальной машине или на сервере (сетевой доступ). В качестве примеров представлены программы, осуществляющие и закрывающие соединения с базой данных, а также выполняющие операторы SQL для обращения к таблицам базы, добавления или обновления строк.

Рассматривались вопросы обработки больших объемов данных и применение курсоров для организации результатов запросов в легко управляемые компоненты. Разговор также шел о блокировках и путях создания приложений, продолжающих обсуживать пользователя во время работы с сервером базы данных.

Доступ к PostgreSQL из C можно получить и с помощью встроенного SQL, который является темой следующей главы.

14

Доступ к PostgreSQL из C при помощи встроенного SQL

В предыдущей главе был рассмотрен общепринятый способ создания клиентских приложений PostgreSQL – написание на языке C программы, взаимодействующей с базой данных. Была использована библиотека `libpq`, содержащая набор специальных функций для PostgreSQL, которая позволяет программе соединиться с базой данных и производить выборку данных из таблиц. Была показана возможность эффективного выполнения стандартных запросов SQL, обновления, вставки и удаления строк таблиц базы данных.

Библиотека `libpq` предоставляет нашим приложениям всю мощь СУБД PostgreSQL, но плохо то, что сам интерфейс с базой данных представляет собой, если можно так выразиться, частную собственность. Другими словами, он предназначен только для PostgreSQL. К тому же в нем нелегко увидеть применяемые операторы SQL, т. к. вспомогательный код имеет тенденцию к скрытию всех важных операторов SQL.

Но помощь уже совсем близко. Многие СУБД, особенно коммерческие, поддерживают встроенный SQL. Стандарт SQL92 определяет интерфейсы для встроенного SQL в другие языки, причем не только для C, но и для FORTRAN, ADA и некоторых других. В декабре 1998 года ANSI также утвердил стандарт для SQL, встроенного в Java, – SQLJ.

PostgreSQL также предоставляет транслятор, преобразующий SQL, который при написании был встроен в код на C, в вызовы `libpq`. Проще говоря, этот способ позволяет помещать операторы SQL в программу на C вместо непосредственного вызова функций `libpq`. Oracle и Informix имеют подобные трансляторы для PRO*C и ESQL/C соответствен-

но, как и многие другие реляционные СУБД. Эквивалент PostgreSQL, ecpg, достаточно близко соответствует стандарту ANSI.

Транслятор, или, строго говоря, препроцессор, работает во многом аналогично препроцессору C. Он считывает программный файл (условимся использовать расширение pgc для PostgreSQL) и порождает файл программы на C, который может быть обработан компилятором. Встроенный SQL заменяется вызовами библиотечных функций ecpg, которые в свою очередь вызывают библиотечные функции libpq.

Применение встроенного SQL и следование стандартам для встроенного в C SQL позволяет создавать приложения, обладающие большей переносимостью на другие базы данных. Также появляется возможность более простой передачи сведений из одной базы данных в другую. Тем, кто имел дело со встроенным SQL в другой СУБД, многое из этой главы уже знакомо.

Первая программа с использованием встроенного SQL

Препроцессор PostgreSQL для встроенного SQL, ecpg, достаточно близко следует стандарту ANSI. Прежде чем приступить собственно к теме данной главы, давайте создадим (при помощи psql) базу данных test с таблицей number, над которой и будем производить дальнейшие эксперименты аналогично тому, как это было в предыдущей главе. Те, кто уже удалил базу данных test, с которой мы работали в предыдущей главе, могут пропустить первый шаг – удаление базы посредством DROP DATABASE:

```
bpsimple=# DROP database test;
DROP DATABASE
bpsimple=# CREATE database test;
CREATE DATABASE
bpsimple=# \c test
You are now connected to database test.
test=# create table number (
test(# intval integer,
test(# name varchar
test(# );
CREATE
test=# INSERT INTO number(intval, name) VALUES(42, 'six times seven');
INSERT 19107 1
test=# INSERT INTO number(intval, name) VALUES(1, 'Numero Uno');
INSERT 19231 1
test=# INSERT INTO number(intval, name) VALUES(111, 'Nelson ');
INSERT 19253 1
test=#
```

Попробуйте сами. Первая программа со встроенным SQL

Теперь рассмотрим очень простой пример применения `esqlc`, программу `update.pgc`:

```
int main()
{
  EXEC SQL connect to test;
  EXEC SQL UPDATE number
           SET name = 'The Answer to the Ultimate Question'
           WHERE intval = 42;
  EXEC SQL commit work;
  EXEC SQL disconnect all;
  return 0;
}
```

Как это работает

Эта программа просто подключается к базе данных (предполагается, что она находится на локальной машине) и обновляет одну из строк таблицы `number`.

> ***Имя базы данных не заключается в кавычки. В более ранних, чем 2.1.0, версиях*** `ecpg` ***оно должно было находиться в одинарных кавычках, поэтому в старых программах можно встретить такую форму записи. Если же использовать эту форму записи в более новых версиях*** `ecpg`***, то программа не будет выполнена, а PostgreSQL зарегистрирует ошибку и выдаст сообщение о том, что база данных*** `'test'` ***не существует.***

Как видите, несколько операторов SQL просто вставлены внутрь главной программы. Синтаксис встроенного SQL достаточно прост: каждый оператор SQL, который должен быть транслирован, должен предваряться строкой `exec sql` и завершаться точкой с запятой:

```
EXEC SQL <некий оператор SQL>;
```

Ключевые слова встроенного SQL нечувствительны к регистру (в том числе и `EXEC SQL`), поэтому можно вводить их как в верхнем, так и в нижнем регистре. Некоторым разработчикам нравится верхний регистр, т. к. SQL выделяется на фоне окружающего C-кода, другие же считают это некрасивым и выбирают нижний регистр. Конечно, следует придерживаться какого-то одного стиля. Мы же будем писать все ключевые слова SQL в верхнем регистре, т. к. это поможет выделить их из кода, написанного на C. Имена переменных встроенного SQL чувствительны к регистру и должны соответствовать своим объявлениям.

Следующим этапом будет применение транслятора для создания файла на C, который можно будет компилировать. Можно оттранслиро-

вать программу из примера, запустив ecpg и указав эту программу в качестве аргумента:

```
$ ecpg -t update.pgc
$
```

Для того чтобы сообщить ecpg, где искать дополнительные заголовочные файлы, добавьте аргумент командной строки -I<каталог включаемых файлов>.

Теперь у нас есть файл программы на C, update.c, который содержит преобразованный исходный текст нашей программы. Те, кому это любопытно, могут посмотреть на несколько «упорядоченный» результат трансляции update.pgc:

```
#include <ecpgtype.h>
#include <ecpglib.h>
#include <ecpgerrno.h>

main()
{
  { ECPGconnect(__LINE__, "test" , NULL, NULL , NULL, 1); }
  { ECPGdo(__LINE__, NULL, "UPDATE  number SET name  = 'The Answer to the
    Ultimate Question'  WHERE intval  = 42", ECPGt_EOIT, ECPGt_EORT);}
  { ECPGtrans(__LINE__, NULL, "commit");}
  { ECPGdisconnect(__LINE__, "ALL");}
}
```

Как видите, операторы SQL были заменены вызовами функций. Все эти функции ecpg имеют названия, начинающиеся с ecpg, и напоминают функции libpq, изученные в предыдущей главе. Функции ecpg доступны в виде библиотеки и используют библиотеку libpq.

Для создания исполняемого файла необходимо скомпилировать update.c и скомпоновать его с обеими библиотеками (ecpg и libpq). В зависимости от того, как была установлена PostgreSQL, каталог, указываемый в параметре -L, может отличаться от нашего:

```
$ cc -o update update.c -L/usr/local/pgsql/lib -lecpg -lpq
$
```

Может также понадобиться задание аргумента командной строки -I<каталог> для указания местоположения включаемых файлов ECPG.

Теперь программа при запуске соединяется с базой данных, обновляет строку и отсоединяется:

```
$ ./update
$
```

Смотреть не на что, т. к. программа не порождает никакого вывода. Проверить, корректно ли произведено обновление, можно при помощи psql.

Если при попытке запуска программы выдается сообщение об ошибке, говорящее о том, что файл разделяемой библиотеки libecpg.so.<некоторое число> не существует, то нужно установить переменную LD_LIBRARY_PATH так, чтобы она включала в себя каталог, содержащий данный файл. Например:

```
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/pgsql/lib
$ export LD_LIBRARY_PATH
$

$ psql -d test
test=# SELECT * FROM number;
  intval |                name
---------+-----------------------------------
       1 | Numero Uno
     111 | Nelson
      42 | The Answer to the Ultimate Question
(3 rows)
test=# \q
$
```

По умолчанию ecpg работает с базой данных так, чтобы операторы выполнялись внутри открытой транзакции, но она не завершает транзакцию автоматически. В конце программы необходимо завершить транзакцию явно, выполнив COMMIT, если вы удовлетворены всеми сделанными в базе данных изменениями. Чуть позже будет рассказано о том, как получить полный контроль над обработкой транзакций.

Чтобы упростить процесс обращения к транслятору и сборку встроенных приложений SQL, используем make-файл, очень похожий на свой аналог из предыдущий главы. Вот make-файл, применимый для программ ecpg:

```
# Make-файл для программ-примеров
# из основ PostgreSQL

# Отредактируйте основные каталоги для
# вашей инсталляции PostgreSQL

INC=/usr/local/pgsql/include
LIB=/usr/local/pgsql/lib

CFLAGS=-I$(INC)
LDLIBS=-L$(LIB) -lecpg -lpq
ECPGFLAGS=-t -I$(INC)
```

```
.SUFFIXES: .pgc
.pgc.c:
    ecpg $(ECPGFLAGS) $<

ALL= cursor insert insert2 select select2 select3 update update2
```

В этом make-файле говорится, что для создания файла вида `.c` из файла `.pgc` надо запустить `ecpg` для файла `.pgc`. Устанавливаем переменную `ECPGFLAGS` для передачи любых необходимых аргументов командной строки транслятору. В данном случае был указан параметр `-t` для обеспечения ручного управления транзакциями. Правила, действующие по умолчанию, заботятся о компиляции и компоновке, так что остается только добавить библиотеку `ecpg` в переменную `LDLIBS`.

Более подробная информация о make-файлах представлена в книге «Beginning Linux Programming» (Основы программирования в Linux) Нейла Мэттью (Neil Matthew) и Ричарда Стоунза (Richard Stones) (Wrox Press, ISBN 1-861-00297-1), а также в руководстве GNU по утилите make, доступном на веб-сайте *http://www.gnu.org*.

Используя make-файл, можно наблюдать шаги, предпринятые при создании программы:

```
$ rm -f update.c update
$ make update
ecpg -t -I/usr/local/pgsql/include update.pgc
cc -I/usr/local/pgsql/include -c -o update.o update.c
cc update.o -L/usr/local/pgsql/lib -lecpg -lpq-o update
rm update.o update.c
$
```

Промежуточный файл `update.c` и исполняемый `update` были удалены, чтобы заставить `make` выполнить все необходимые для построения программы шаги. Отметьте также, что (по крайней мере, в версии GNU) `make` удаляет промежуточный и объектный файлы, чтобы сохранить все в чистоте. В итоге остается только `update.pgc`, исходный текст встроенной SQL-программы и исполняемый двоичный файл, `update`.

Аргументы ecpg

Для управления поведением `ecpg` можно использовать некоторые аргументы командной строки. Полная команда будет выглядеть следующим образом:

```
ecpg [-v] [-t] [-I include-dir] [-o output-file] file1 [file2 …]
```

Описание аргументов представлено в табл. 14.1.

Таблица 14.1. Аргументы ecpg

Аргумент	Результат
`-v`	Выводит информацию о версии в стандартный вывод ошибок `stderr`.
`-t`	Выключает режим автоматических транзакций, в котором каждый отдельный оператор автоматически выполняется внутри транзакции. Для закрытия транзакции необходим `COMMIT`.
`-o`	Задает имя выходного файла для обработанного программного кода. По умолчанию имеет такое же имя, что и входной файл, а расширение `.pgc.` заменяется на `.c`.
`-I`	Добавляет указанный каталог в список каталогов, просматриваемых в поиске заголовочных файлов, включенных в исходный файл.

Аргумент `-t` регулирует способ применения транзакций. По умолчанию (без `-t`) `ecpg` делает так, что все операторы выполняются внутри транзакции, как будто непосредственно перед ними написано `BEGIN WORK`. Поскольку PostgreSQL не поддерживает вложенные транзакции, все операторы выполняются в одной транзакции. Для того чтобы зафиксировать изменения, явно введите оператор `COMMIT`. Можно сделать это сколько угодно раз. Следующий оператор SQL автоматически начнет другую транзакцию.

Если вы хотите сами контролировать транзакции в программе (обычно это разумно), рекомендуем указать аргумент `-t` для `ecpg` и явно использовать SQL-операторы `BEGIN WORK` и `COMMIT WORK`.

> ***Оставшиеся примеры данной главы будут работать корректно, только если они обрабатываются*** `ecpg` ***с аргументом*** `-t` ***или же если используется представленный ранее Makefile.***

В примере make-файла был явно указан путь поиска для заголовочных файлов (точно так же мы поступали в предыдущей главе при компиляции приложений `libpq`) и параметр для ручного управления транзакциями.

Журнал выполнения SQL

Многие библиотечные функции `ecpg` вызывают внутри себя функцию записи в журнал в целях отладки. К этой возможности можно прибегнуть для генерирования журнала выполнения операторов SQL нашей программой со встроенным SQL. Для этого разрешаем вывод отладочных сообщений вызовом `ECPGdebug`:

```
void ECPGdebug(int logging, FILE *logstream);
```

Передаем любое ненулевое значение в качестве параметра `logging`, чтобы разрешить отладочный вывод, и нулевое значение, чтобы запретить его. Информация будет выводиться в выходной поток, указанный параметром `logstream`.

Добавьте приведенную ниже строку в начало функции `main` в `update.pgc` и пересоберите:

```
ECPGdebug(1,stderr);
```

Теперь можно посмотреть на процесс выполнения программы:

```
$ make update
$ ./update
[1597]: ECPGdebug: set to 1
[1597]: ECPGconnect: opening database test
[1597]: ECPGexecute line 15: QUERY: update number set name  = 'The Answer to
the Ultimate Question' where intval  = 42 on connection test
[1597]: ECPGexecute line 15 Ok: UPDATE 1
[1597]: ECPGtrans line 16 action = commit connection = test
[1597]: ecpg_finish: finishing test.
```

Число, указанное в скобках в начале каждой строки отладки, представляет собой идентификатор процесса программы и будет изменяться при каждом перезапуске. Это удобно для разделения вывода различных программ, работающих в одно и то же время.

Функция `ECPGdebug` (и другие явно вызываемые функции `ECPG`) специфична для PostgreSQL, поэтому приложение, в котором она применяется, не обладает переносимостью на другие реализации баз данных, поддерживающие встроенный SQL. Вероятно, применение этих функций лучше ограничить этапами тестирования и отладки приложения, а затем удалить их или же включать только при помощи директив условной компиляции препроцессора C.

Соединения с базой данных

В представленном ранее примере программы `update.pgc` содержится весьма упрощенный вариант оператора SQL для соединения с базой данных. Практически все параметры, которые могли быть заданы в эквивалентных функциях `libpq`, были установлены в их значения по умолчанию.

Приведенный ниже оператор реализует попытку соединения с базой данных `test`, находящейся на локальной машине, подставляя регистрационное имя текущего пользователя и не указывая пароль:

```
EXEC SQL connect to test;
```

Применив полную версию оператора `connect`, можно указать больше деталей: к какому серверу подключаться, какой порт прослушивает сервер базы данных, а также имя пользователя и пароль:

```
EXEC SQL CONNECT database_url
    AS connection_name
    USER login_name
    USING password;
```

Расположение базы данных и способ соединения с ней определяет параметр database_url (подобно тому, как URL в Интернете определяет местоположение файла или веб-страницы), который может быть указан в одной из форм, перечисленных в табл. 14.2:

Таблица 14.2. Варианты указания database_url

Форма	Описание
database_name	База данных на локальной машине
database_name@server	База данных на удаленном сервере
database_name@server:port	Удаленная база данных и нестандартный порт
tcp:postgresql://server	База данных по умолчанию на удаленном сервере, соединенная через сокет TCP
tcp:postgresql://server:port	Удаленная база данных и нестандартный порт
tcp:postgresql://server/database_name	Удаленная база данных с указанным именем
unix:postgresql://server	База данных, соединенная через сокет домена UNIX
default	База данных по умолчанию, определяемая переменной окружения PGDATABASE
:host_variable	Информация о соединении получается из строковой переменной C

Для идентификации соединения в последующих операторах SQL применяется идентификатор connection_name.

Имя пользователя – это login_name. Пароль может быть добавлен после имени пользователя (отделяется косой чертой).

Наконец, password – это пароль пользователя. Ключевое слово identifiedby может применяться в операторе connect как синоним для using при указании паролей. Сведения о пользователе при необходимости могут быть получены из переменной. Подробности применения переменных в операторах SQL приведены в разделе «Переменные основного языка» далее в этой главе.

Если вы пишете программу, в которой осуществляется несколько соединений, то присваивание им имен позволит идентифицировать их в операторах SQL. Чтобы указать определенное соединение с базой данных, применим расширенную форму exec sql:

```
EXEC SQL AT connection_name <sql statement>;
```

Также можно изменить соединение, используемое в операторах SQL (текущее соединение), с помощью оператора SET:

```
EXEC SQL SET CONNECTION TO connection_name;
```

Соединение с базой данных закрывается оператором DISCONNECT:

```
EXEC SQL DISCONNECT connection;
```

Его необязательный аргумент connection может иметь значения, представленные в табл. 14.3:

Таблица 14.3. Значения аргумента оператора DISCONNECT

Вариант	Описание
Default	Соединение по умолчанию
Current	Текущее соединение
All	Все соединения
connection_name	Указанное соединение

В некоторых случаях полезной может оказаться функция ECPGstatus, входящая в библиотеку ecpg, которая возвращает ненулевое булево значение (TRUE), если соединение с базой данных действительно (т. е. установлено), и FALSE – в ином случае:

```
bool ECPGstatus(int lineno, char *connection_name);
```

Параметры ECPGstatus следуют стандарту функций ECPG, и первым параметром является номер строки (обычно используется макрос __LINE__ препроцессора C). Второй параметр – это строка, предназначенная для идентификации рассматриваемого соединения. Можно употребить тот же дескриптор или имя соединения, что применялись во встроенном операторе SQL connect при установлении соединения.

Чтобы проверить, успешно ли наша программа подключилась к базе данных, добавим в нее такой тест:

```
if(!ECPGstatus(__LINE__,"test")) {
    /* failed to connect */
}
```

Опять-таки, применение ECPGstatus сделает программу специфичной для PostgreSQL. Как мы увидим в дальнейшем, есть более стандартный способ проверки успешности соединения, но он работает лишь сразу после осуществления попытки подключения к базе.

Обработка ошибок

Рассматриваемая в качестве примера программа сейчас действует достаточно бесцеремонно. Она продолжается невзирая ни на что, не обращая внимания на любые ошибки, которые могут возникнуть при выполнении одного из встроенных операторов SQL. По существу, может случиться, что программа зависнет, если ей не удастся соединиться с базой данных, а она будет пытаться производить обновления.

Будучи ответственными разработчиками, мы должны обнаружить возможные ошибки и исправить ситуацию.

Стандартный встроенный SQL определяет механизм выдачи сообщений об ошибках, и PostgreSQL поддерживает его. Стандартная структура (управляющий блок SQL) определена и называется `sqlca`. На C она может быть определена следующим образом:

```
struct
{
    char        sqlcaid[8];
    long        sqlabc;
    long        sqlcode;
    struct
    {
        int         sqlerrml;
        char         sqlerrmc[70];
    } sqlerrm;
    char        sqlerrp[8];
    long        sqlerrd[6];
    char        sqlwarn[8];
    char        sqlext[8];
} sqlca;
```

Структура `sqlca` служит для передачи сообщений об ошибках и значений статусов выполнения операторов встроенного SQL Она довольно загадочна, а различные коды могут показаться странными. Кроме того, PostgreSQL обеспечивает не всю информацию, которая может передаваться через поля `sqlca`. Коммерческие базы данных, такие как Oracle, предоставляют дополнительную информацию, но и приведенная реализация также вполне пригодна к употреблению. Здесь описаны только поля, реализованные в PostgreSQL.

Структура `sqlca` возвращается в исходное состояние после каждого оператора встроенного SQL, так что информацию, необходимую для какого-либо оператора, необходимо извлечь до выполнения следующего. Поля структуры устанавливаются в зависимости от того, что именно происходит, и ключевым является поле `sqlcode`.

Поле `sqlca.sqlcode` будет установлено в код результата, который равен нулю, если все прошло успешно. Для серьезных ошибок возвращается отрицательная величина. Например, если попытка соединения с базой данных не удалась, то `sqlca.sqlcode` будет установлено в `-402`. Исправимые ошибки возвращают положительные значения. Чрезвычайно важным положительным результатом представляется код `100`, означающий, что оператор `SELECT` не возвратил данных, что не является ошибкой.

Если возникла ошибка, то строка `sqlca.sqlerrm.sqlerrmc` будет содержать сообщение, эту ошибку описывающее. Длина сообщения (которое заканчивается нулевым байтом) задается в `sqlca.sqlerrm.sqlerrml`. В PostgreSQL сообщения об ошибках могут показаться не очень полез-

ными, т. к. вместо чего-то содержательного они могут представлять собой нечто вроде "error #-203". Коды ошибок и их значения приведены в табл. 14.4:

Таблица 14.4. Ошибки выполнения встроенных операторов SQL

Код	Описания
-12	Нехватка памяти.
-201	Слишком много аргументов. Вероятно, вызвано тем, что PostgreSQL возвратила больше значений, чем программа разрешает для результата выполнения оператора SELECT.
-202	Слишком мало аргументов. Вероятно, вызвано тем, что PostgreSQL возвратила меньше данных, чем программа ожидала.
-203	Слишком много совпадений. Запрос вернул несколько строк, а переменные, получающие эти данные, достаточны только для одной строки.
-208	Пустой запрос. Эквивалентно статусу PGRES_EMPTY_QUERY, возвращаемому libpq, и, вероятно, указывает на ошибку программы, т. к. сервер попросили не работать.
-220	Нет такого соединения. Программа назвала соединение, которое не существует.
-221	Нет соединения. Программа попыталась получить доступ к данным, не установив соединение.
-400	Ошибка PostgreSQL. В сообщении будут содержаться подробности об ошибке на сервере.
-401	Ошибка транзакции. Возникла ошибка при открытии, фиксации или откате транзакции.
-402	Ошибка открытия. Невозможно установить соединение с базой данных.
100	Не найдено. Запрос не возвратил данных.

После успешного выполнения оператора INSERT, UPDATE или DELETE количество строк, на которые было оказано воздействие, появляется в sqlca.sqlerrd[2]. Это поле также используется при возврате данных посредством SELECT и с применением курсоров.

В некоторых случаях бывает так, что, хотя ошибка не является фатальной, она должна быть донесена до сведения приложения. Для исправимых ошибок и в некоторых других ситуациях выдаются предупреждения. Когда данные возвращаются в результате выполнения оператора SQL, такого как SELECT, можно сделать так, чтобы они принимались переменными C (об этом будет рассказано чуть позже в этой главе). Если данных слишком много для переменной, то они отсекаются и выдается предупреждение.

Массив sqlca.sqlwarn служит для передачи информации о предупреждениях: sqlca.sqlwarn[0] будет установлен в W, если было выдано предупреждение, sqlca.sqlwarn[1] будет установлен в W, если данные были

отсечены при получении в переменную C, sqlca.sqlwarn[2] будет установлен в W, если произойдет исправимая ошибка.

Мы можем использовать структуру sqlca в нашей программе. Для этого надо сказать ecpg, чтобы она включила в проект ее определение. Применим такую директиву:

```
EXEC SQL INCLUDE sqlca;
```

Попробуйте сами. Обновления

Создадим вторую программу обновления, которая будет проверять количество измененных строк. Эта версия (update2.pgc) изменяет себестоимость и цену продажи товара в основной учебной базе данных, bpsimple:

```
#include <stdio.h>

EXEC SQL include sqlca;

main()
{
  ECPGdebug(1,stderr);

  EXEC SQL connect to bpsimple;

  EXEC SQL UPDATE item
           SET cost_price = 1.75, sell_price = 2.99
           WHERE description = 'Linux CD';

  if(sqlca.sqlcode == 0)
    printf("rows affected: %d\n", sqlca.sqlerrd[2]);

  EXEC SQL disconnect all;
}
```

Как это работает

Программа извлекает количество строк, измененных оператором UPDATE, если он был выполнен. Нулевой код в sqlca.sqlcode означает, что UPDATE отработал успешно. Количество измененных строк представлено в sqlca.sqlerrd[2].

Обработчики ошибок

Можно сделать так, чтобы наша ecpg-программа автоматически выполняла некоторый код всякий раз, встретив ошибку или предупреждение. Можно также принять меры, чтобы код выполнялся каждый раз, когда имеет место псевдоошибка (отсутствие данных) с номером 100.

Используем оператор whenever:

```
EXEC SQL whenever condition action;
```

Возможные условия (condition) приведены в табл. 14.5:

Таблица 14.5. Варианты условий для оператора whenever

Условие	Описание
Sqlerror	Произошла фатальная ошибка
Sqlwarning	Произошла нефатальная ошибка
Not found	Данные не возвращены

Действие же (action), вызванное наступлением того или иного условия, указано в табл. 14.6:

Таблица 14.6. Варианты действий для оператора whenever

Действие	Описание
Sqlprint	Вывести сообщение об ошибке на стандартный вывод stderr
do c_code	Выполнить функцию C

Если вас устроит вывод сообщений для исправимых ошибок и выход из программы при наличии фатальной ошибки, то можно включить в приложение следующий код для перехвата и обработки ошибок:

```
EXEC SQL whenever sqlwarning sqlprint;
EXEC SQL whenever sqlerror do GiveUp();

void GiveUp()
{
    fprintf(stderr, "Fatal error\n");
    sqlprint();
    exit(1);
}
```

PostgreSQL реализует в ecpg оператор whenever, генерируя код для проверки значений sqlca.sqlcode и sqlca.sqlwarn[0] после каждого встроенного оператора SQL и выполняя определенное действие, если проверка не удалась. Ключевое слово sqlprint – это сокращение от do sqlprint(), инициирующего вызов библиотечной функции, выводящей соответствующее сообщение об ошибке.

Функции whenever специфичны для PostgreSQL и не доступны в стандартном встроенном SQL. Поэтому если существует какая-то вероятность, что впоследствии программа будет переноситься на другую базу данных, не применяйте их в конечном коде.

Переменные основного языка

Пока что в нашей программе-примере просто выполнялись постоянные операторы SQL, и вас наверняка интересует, как ввести какие-то

переменные данные. Ведь применяя libpq, можно создать оператор SQL в символьной строке (с помощью sprintf) и таким образом добиться того, чтобы он содержал данные, полученные из переменных.

Встроенный SQL также допускает наличие переменных в своих операторах, причем его способ гораздо проще, чем манипулирование символьными строками, которым занимается libpq. Чтобы сослаться на переменную во встроенном SQL, необходимо указать ее имя, предваренное двоеточием. Вставить новую строку в таблицу item можно следующим образом:

```
EXEC SQL INSERT INTO item VALUES(:description, :cost_price, :sell_price);
```

Обратите внимание, что значение для item_id можно опустить, т. к. оно автоматически генерируется при добавлении строки в таблицу item. К переменным (description, cost_price и sell_price) обращаемся как к переменным основного языка, т. к. это переменные, содержащиеся в клиентском приложении, а не на сервере базы данных.

Для того чтобы работать в своей программе с переменными C (то есть основного языка), необходимо оповестить о них ecpg. Делаем это, объявляя переменные, которые будут использоваться в операторах SQL, в специальных разделах объявлений:

```
EXEC SQL BEGIN declare section;
Здесь объявляются переменные основного языка
. . . .
EXEC SQL END declare section;
```

Разделы объявлений должны располагаться там, где разрешено объявлять переменные C, иными словами – в начале блока или вне функций. Дело в том, что они будут переработаны в обычные объявления переменных C, а также зарегистрированы ecpg как переменные основного языка.

Для простых значений, таких как целые числа и строки фиксированной длины, можно объявлять переменные основного языка так же, как в C.

Попробуйте сами. Переменные основного языка

Начнем с написания программы (insert.pgc), которая позволит добавлять новые штрих-коды товаров в нашу базу данных. Идентификатор продукта и штрих-код будут параметрами командной строки:

```
#include <stdio.h>
#include <string.h>

EXEC SQL include sqlca;
EXEC SQL whenever sqlwarning sqlprint;
```

```
EXEC SQL whenever sqlerror do GiveUp();

void GiveUp()
{
    fprintf(stderr, "Fatal Error\n");
    sqlprint();
}

main(int argc, char *argv[])
{
  EXEC SQL begin declare section;
  int item_id;
  char barcode[13];
  EXEC SQL end declare section;

  if(argc != 3) {
    printf("usage: INSERT item barcode\n");
    exit(1);
  }

  item_id = atoi(argv[1]);
  strncpy(barcode, argv[2], sizeof(barcode));

  EXEC SQL connect to bpsimple;
  EXEC SQL insert into barcode values(:barcode, :item_id);
  EXEC SQL disconnect all;
}
```

Как это работает

Были объявлены две переменные основного языка, `item_id` и `barcode`, соответствующие типам данных (тип целых чисел и строк фиксированной длины) для столбцов таблицы продуктов. Ими можно оперировать в нашей программе на C, потому что это обычные переменные C. Можно использовать их во встроенном SQL, т. к. они были объявлены для `ecpg` внутри раздела объявлений. Обращаясь к ним в SQL, просто ставим перед их именами двоеточие.

Скомпилировав и запустив программу, добавим с ее помощью в базу данных новые штрих-коды. При попытке ввести дублирующий штрих-код возникнет фатальная ошибка, которая будет обработана конструкцией `sql whenever`:

```
$ make insert
$ ./insert 2 1234567890123
$ ./insert 2 1234567890123
Fatal Error
sql error 'ERROR:  Cannot insert a duplicate key into unique index
barcode_pk'
$
```

Если требуется работать с другими типами данных для атрибутов, то их следует объявить. Что касается чисел с плавающей точкой для обработки цен, хранимых в базе данных как `NUMERIC(7,2)`, выберем тип C `double`. Для дат используем символьный тип достаточной длины.

Небольшая трудность возникает, когда мы добираемся до типа базы данных `VARCHAR`. Этот тип содержит переменное количество символов и не обязательно заканчивается нулевым значением, в отличие от обычных символьных строк C. Не существует простого способа представить тип `VARCHAR` в C, поэтому придется прибегнуть к структуре, состоящей из двух элементов: символьного массива максимальной длины и целого счетчика, регистрирующего количество значащих символов в массиве.

К счастью, об объявлениях может позаботится `ecpg`. Нам же остается просто использовать псевдотип `VARCHAR` вместо `char` при объявлении переменной, а `ecpg` объявляет структуру, состоящую из элементов `arr` и `len`, содержащих соответственно символы строки и ее длину.

Попробуйте сами. Данные переменной длины

В следующем примере (`insert2.pgc`) будем рассматривать программу, добавляющую новые товары в таблицу `item` нашей базы данных. Берем описание продукта (`VARCHAR(64)`) и два числа с плавающей точкой, представляющие себестоимость и цену продажи, и вставляем новую строчку в таблицу:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

EXEC SQL include sqlca;
EXEC SQL whenever sqlwarning sqlprint;
EXEC SQL whenever sqlerror do GiveUp();

void GiveUp()
{
    fprintf(stderr, "Fatal Error\n");
    sqlprint();
}

main(int argc, char *argv[])
{
  EXEC SQL begin declare section;
  char dbname[] = "bpsimple";
  double cost_price, sell_price;
  VARCHAR description[64];
  EXEC SQL end declare section;

  if(argc != 4) {
    printf("usage: insert description cost_price sell_price\n");
```

```
    exit(1);
  }

  strncpy(description.arr, argv[1], sizeof(description.arr));
  description.len = strlen(description.arr);
  cost_price = atof(argv[2]);
  sell_price = atof(argv[3]);

  EXEC SQL connect to :dbname as bpsimple;
  EXEC SQL at bpsimple insert into
    item(description, cost_price, sell_price)
    values(:description, :cost_price, :sell_price);

  EXEC SQL disconnect bpsimple;
}
```

Добавить новый продукт можно простым вызовом:

```
$ ./insert2 "Widget" 1.87 2.93
$
```

Проверим, обновлена ли таблица `item`, при помощи `psql`:

```
bpsimple=# SELECT * FROM item;
 item_id |  description  | cost_price | sell_price
---------+---------------+------------+------------
...
      12 | Widget        |       1.87 |       2.93

bpsimple=#
```

Как это работает

Объявляем переменные основного языка в разделе объявлений и используем оба члена структуры описания для создания переменной основного языка, посредством которой в операторе SQL можно организовать вставку новой строки.

Поскольку в таблице `item` столбец `item_id` принадлежит к типу `serial`, то можно (и нужно) опустить `item_id` при вводе новой строки в эту таблицу. Поэтому явно укажем названия столбцов, для которых предоставляются данные, в операторе `INSERT`.

> ***Если требуется вставить значение*** `NULL` ***в столбец строки таблицы, можно использовать ключевое слово null в инструкции values оператора*** `INSERT`.

В этой программе, ради разнообразия, для задания базы данных, с которой необходимо установить соединение, и демонстрации использования указанного соединения, применяется переменная основного языка.

Извлечение данных с помощью ecpg

После введения переменных основного языка можно приступать к рассмотрению извлечения данных из базы, т. к. переменные основного

языка служат хранилищем для кортежей (строк), возвращенных оператором SELECT. Начнем с очень простого примера. Подсчитаем количество наших клиентов:

```
EXEC SQL begin declare section;
int count;
EXEC SQL end declare section;

EXEC SQL SELECT count(*) INTO :count FROM customer;
```

Оператор SELECT дополнен инструкцией INTO, задающей переменные основного языка, которые должны применяться для извлечения информации. В этом примере была объявлена целая переменная count, которая в случае успешного выполнения данного оператора SQL будет содержать количество клиентов, хранящихся в базе данных.

Аналогично можно извлечь строку данных в переменные, указав в инструкции INTO список переменных основного языка, соответствующих выбираемым данным. Чтобы извлечь подробные сведения о конкретном клиенте по его customer_id, напишем:

```
EXEC SQL SELECT addressline, zipcode INTO :addr, :zip
         FROM customer
         WHERE customer_id = 15;
```

Вместо того чтобы использовать "*", всегда следует явно перечислять в операторах SELECT выбираемые столбцы. Если в схему базы данных будут вноситься изменения, то код будет гораздо более устойчивым, если же его все-таки не удастся выполнить, т. к. упомянутый столбец окажется удаленным, то, по крайней мере, будет выдано содержательное сообщение об ошибке.

Переменные основного языка можно использовать и в других инструкциях операторов SELECT. Чтобы получить customer_id из переменной основного языка, напишем:

```
WHERE customer_id = :id;
```

В учебной базе данных столбец customer_id таблицы customer принадлежит к типу serial, поэтому клиент, добавленный последним, всегда имеет максимальный идентификатор в таблице.

Попробуйте сами. Извлечение данных

Данная программа находит максимальный customer_id и по нему извлекает почтовый код и первую строку адреса для этого клиента:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
EXEC SQL include sqlca;
EXEC SQL whenever sqlwarning sqlprint;
EXEC SQL whenever sqlerror do GiveUp();

void GiveUp()
{
    fprintf(stderr, "Fatal Error\n");
    sqlprint();
}

main(int argc, char *argv[])
{
  EXEC SQL begin declare section;
  int id;
  char zip[10];
  VARCHAR address[64];
  int address_ind;
  EXEC SQL end declare section;
  EXEC SQL connect to bpsimple;
  EXEC SQL SELECT max(customer_id) INTO :id FROM customer;
  printf("We have %d customers\n", id);

  EXEC SQL SELECT addressline, zipcode
           INTO :address:address_ind, :zip
           FROM customer
           WHERE customer_id = :id;

  printf("The address is%sNULL\n", address_ind ? " " : " not ");
  printf("customer id: %d\n", id);
  printf("%.*s <%.*s>\n", sizeof(zip), zip, address.len, address.arr);

  EXEC SQL disconnect all;
}
```

select.pgc

Скомпилировав и запустив программу, получим представленный ниже вывод:

```
$ ./select
We have 15 customers
The address is not NULL
customer id: 15
MT2 6RT  <4 The Square>
```

Как это работает

Программа извлекает данные, возвращенные оператором SELECT, в несколько переменных основного языка и, поступая таким образом, обнажает несколько подводных камней. Их наличие является результа-

том некоторых тонкостей программы, которые следует очень аккуратно продумать.

Прежде всего, мы манипулируем символьными данными из базы данных, извлеченными в строки C. Обычно строки C завершаются нулевым значением, и многие библиотечные функции предполагают его наличие. Например, если просто ввести строку, то функция `printf` будет выводить символы до тех пор, пока не встретит нулевое значение в строке. Потенциально еще хуже случай копирования символьных данных, т. к. `strcpy` будет копировать вслепую, пока не встретит нулевое значение.

Символьные данные в базе данных не обязательно завершаются нулем, в связи с чем данные из столбца, возвращаемые оператором `SELECT`, могут создать проблему. Выводя их при помощи `printf`, мы можем увидеть ненужную информацию, т. к. `printf` выводит символы из области памяти, которая превышает размеры наших данных, пока не встретит нулевое значение. Копирование посредством `strcpy` будет продолжаться до тех пор, пока не встретится нулевое значение, а это может произойти и за пределами символьных данных, что приведет к искажению данных, следующих за строкой назначения.

Одним из способов решения этой проблемы может быть соблюдение особой осторожности при копировании данных для обработки их в программе: необходимо учесть дополнительное пространство для нулевого значения и добавить его в конец строки.

Наша программа использует свойство функции `printf` ограничивать объем выводимых символьных данных. Поле точности для формата `%s` позволяет указать максимальный объем вывода символьных данных.

Представленная ниже строка выведет 10 символов почтового индекса фиксированной длины, не требуя нулевого символа, и не более 64 символов первой строки адреса:

```
printf("%.10s %.64s\n", zip, address.arr);
```

На самом деле в данных переменной длины обычно есть нулевой символ, если только они не имеют максимальную длину.

К выводу символьных данных можно применить чуть более общий подход, задав поле точности в качестве аргумента `printf`, тогда при изменении проекта базы данных программа будет чуть более устойчива. Вот версия вызова `printf`, в которой нет явных длин строк:

```
printf("%.*s %.*s\n", sizeof(zip), zip, address.len, address.arr);
```

Вторая тонкость, на которую необходимо обратить внимание, – это наши старые знакомые – неизвестные значения базы данных. Может случиться, что клиент не сообщил нам свой адрес и столбец содержит `NULL`.

При извлечении данных в переменные основного языка нет простого способа отличить пустую строку от значения NULL или нулевое целое значение от NULL. Подобная ситуация рассматривалась в предыдущей главе. В libpq существует функция PQgetisnull, которая сообщает, является ли извлеченный результат значением NULL. Во встроенном SQL для того, чтобы узнать, является ли возвращенное значение NULL, следует использовать **индикаторные переменные**.

Индикаторная переменная – это целая переменная основного языка, которая используется для того, чтобы показать, является ли извлеченное значение столбца значением NULL. Индикатор указывается вместе с переменной основного языка, которой он сопоставлен. Индикатор добавляется после имени переменной, от которого он отделяется двоеточием:

```
SELECT addressline
       INTO :address:address_ind FROM customer
       WHERE ...
```

Здесь фигурирует индикаторная переменная address_ind, которая будет установлена в ненулевое значение, если значение, извлеченное в переменную основного языка address, представляет собой NULL. Следует игнорировать любое значение (пустая строка, ноль и т. д.), пересылаемое в переменную, если оно имеет значение NULL.

Разумно называть индикаторные переменные так, чтобы было понятно, с какой переменной они связаны. Предлагаем придерживаться обозначения, примененного в предыдущем примере, то есть добавлять к имени главной переменной суффикс _ind (индикатор).

Транзакции

Реализовать транзакции в ecpg чрезвычайно просто. По аналогии с libpq можно ожидать возможности применять стандартный SQL, и так оно и есть. Соответствующие операторы выглядят следующим образом:

```
EXEC SQL BEGIN WORK;
EXEC SQL COMMIT WORK;
EXEC SQL ROLLBACK;
```

Как уже говорилось, по умолчанию (если не задан параметр -t) ecpg делает так, чтобы каждый встроенный оператор SQL выполнялся внутри транзакции (режим autocommit или режим неявных транзакций). Каждый новый оператор начинает или продолжает транзакцию, а для фиксации изменений применяется COMMIT.

Чтобы получить полный контроль над транзакциями, при обработке pgc-файлов, составляющих приложение, укажите параметр -t для ecpg.

Обработка данных

В примерах этой главы мы тщательно следили за тем, чтобы при извлечении данных получалась ровно одна строка, тогда значения столб-

цов сохранялись в переменных основного языка и с ними можно было работать в программе.

Немного изменив программу `select`, осуществляющую поиск клиентов по почтовому индексу, можно продемонстрировать обработку тех случаев, когда не возвращается ни одной строки.

Попробуйте сами. Обработка случая отсутствия данных

Следующая программа, рассматриваемая в качестве примера (`select2.pgc`), выявляет случаи, когда `SELECT` не возвращает данных:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

EXEC SQL include sqlca;
EXEC SQL whenever sqlwarning sqlprint;
EXEC SQL whenever sqlerror do GiveUp();

void GiveUp()
{
    fprintf(stderr, "Fatal Error\n");
    sqlprint();
    exit(1);
}

main(int argc, char *argv[])
{
  EXEC SQL begin declare section;
  int id;
  char title[4];
  int title_ind;
  char zip[10];
  VARCHAR lname[32];
  VARCHAR town[64];
  int town_ind;
  EXEC SQL end declare section;

  if(argc != 2) {
    printf("Usage: select2 zipcode\n");
    exit(1);
  }

  strncpy(zip, argv[1], sizeof(zip));

  EXEC SQL connect to bpsimple;

  EXEC SQL SELECT customer_id, title, lname, town
           INTO :id, :title:title_ind, :lname, :town:town_ind
           FROM customer
```

```
            WHERE zipcode = :zip;

  if(sqlca.sqlerrd[2] == 0) {
    printf("no customer found\n");
  }
  else {
    printf("title is%sNULL\n", title_ind ? " " : " not ");
    printf("town is%sNULL\n", town_ind ? " " : " not ");
    printf("customer id: %d\n", id);
    printf("%.*s %.*s <%.*s>\n",
       sizeof(title), title,
       lname.len, lname.arr,
       town.len, town.arr);
  }
  EXEC SQL disconnect all;
}
```

Как это работает

В программе использовано то обстоятельство, что управляющий блок SQL будет содержать информацию о количестве возвращенных строк в `sqlca.sqlerrd[2]`. Если оно равно нулю, значит, строки не были найдены. Применим эту программу для реального запроса. Учебная база данных содержит такие записи для клиентов:

```
customer_id  | title |  lname  |   town    |  zipcode
-------------+-------+---------+-----------+-----------
           1 | Miss  | Stones  | Hightown  | NT2 1AQ
           2 | Mr    | Stones  | Lowtown   | LT5 7RA
           4 | Mr    | Matthew | Yuleville | YV67 2WR
           5 | Mr    | Cozens  | Oahenham  | OA3 6QW
           6 | Mr    | Matthew | Nicetown  | NT3 7RT
           7 | Mr    | Stones  | Bingham   | BG4 2WE
           8 | Mrs   | Stones  | Bingham   | BG4 2WE
           9 | Mrs   | Hickman | Histon    | HT3 5EM
          10 | Mr    | Howard  | Tibsville | TB3 7FG
          11 | Mr    | Jones   | Bingham   | BG3 8GD
          12 | Mr    | Neill   | Winersby  | WB3 6GQ
          13 | Mrs   | Hendy   | Oxbridge  | OX2 3HX
          14 | Mr    | O'Neill | Welltown  | WT3 8GM
          15 | Mr    | Hudson  | Milltown  | MT2 6RT
           3 | Miss  | Matthew | Nicetown  | NT2 2TX
```

```
$ make select2
$ ./select2 "NT2 2TX"
title is not NULL
town is not NULL
customer id: 3
Miss Matthew <Nicetown>
$ ./select2 "BG4 2XE"
no customer found
```

```
$ ./select2 "BG4 2WE"
Fatal Error
sql error SQL error #-203 in line 39.
$
```

Если при поиске по почтовому индексу (zipcode) не найдены соответствующие записи, то ни одна строка не возвращается. Запрос распознает такую ситуацию и выводит сообщение. Если клиент с заданным значением zipcode найден, выводится информация о нем.

В третьем случае происходит фатальная ошибка, т. к. выбранный почтовый индекс разделяют несколько клиентов. В примере два клиента имеют один и тот же zipcode. Поскольку невозможно сохранить сведения об обоих клиентах в переменных основного языка, выдается ошибка. Чтобы решить проблему возвращения нескольких строк, следует использовать курсоры, как это делалось в предыдущей главе. Вернемся к курсорам чуть позже.

Если возвращенных данных нет, примените конструкцию exec sql whenever: если случится такая ситуация, будет выполнен определенный код. Посмотрим на следующий фрагмент программы:

```
EXEC SQL whenever not found do break;

do {
  EXEC SQL SELECT customer_id, title, lname, town
    INTO :id, :title:title_ind, :lname, :town:town_ind
    FROM customer
    WHERE zipcode = :zip;

  printf("title is%sNULL\n", title_ind ? " ": " not ");
  printf("town is%sNULL\n", town_ind ? " ": " not ");
  printf("customer id: %d\n", id);
  printf("%.*s %.*s <%.*s>\n",
     sizeof(title), title,
     lname.len, lname.arr,
     town.len, town.arr);
} while(0);
```

Здесь, если запрос не возвращает никаких строк, выполняется оператор C break. Сейчас break имеет силу лишь внутри цикла, поэтому создан цикл, работающий один раз (использован общепринятый в C прием – do{} while(0)), который выполнит код, находящийся в блоке do и не вернется к началу цикла, т. к. условие продолжения ложно. Теперь при запуске программы, в которой применен такой метод, будут лишь отображены сведения о клиенте, если он будет найден.

Если вам кажется, что это чересчур сложно, попробуйте представить, что делает транслятор ecpg в случае наличия конструкции sql whenever. Он преобразует:

```
exec sql whenever not found <некоторый код>;
exec sql <некоторый запрос>;
```

в нечто типа:

```
ECPGdo(… <некоторый запрос> …);
if(sqlca.sqlcode == ECPG_NOT_FOUND) <некоторый код>;
```

Код, указанный в операторе `sql whenever`, должен быть действителен в том контексте, в котором он используется транслятором. Поэтому необходимо создать искусственный цикл для `break`, чтобы избежать вывода информации о клиентах, если ничего не было возвращено. При желании можно даже поместить в часть «действие» оператора `sql whenever` вызов функции, в которой определены локальные переменные, поскольку, где бы транслятор ни использовал код, переменная будет доступна.

Неумеренное применение конструкции `sql whenever` может привести к тому, что за кодом будет тяжело проследить, поэтому некоторые разработчики заявляют, что использовать ее так же плохо, как и `goto`. Вероятно, не стоит слишком увлекаться `sql whenever` и лучше задавать простой код в качестве выполняемого действия, чтобы облегчить восприятие приложения.

Вообще, лучшим способом гарантировать возвращение ровно одной строки являются курсоры, которыми мы сейчас и займемся.

Курсоры

Последней темой обсуждения данной главы является реализация курсоров во встроенном SQL. В предыдущей главе курсоры уже были представлены как превосходное универсальное средство обслуживания неизвестного количества возвращаемых строк.

Для тех, кто пропустил главу 13, повторим приведенный там совет. Вообще говоря, следует избегать написания программ, которые предполагают, что оператор `SELECT` возвращает одну строку или не возвращает ни одной строки; если только это не простые агрегатные функции, такие как запрос `SELECT COUNT(*) FROM` или `SELECT` для первичного ключа, когда можно гарантировать, что результатом будет ровно одна строка. Если же есть сомнения, применяйте **курсор**.

Чтобы разобраться с множеством строк, возвращаемых запросом, будем извлекать их по одной, используя `FETCH`, получая значения столбцов в переменные основного языка так же, как это было с одной возвращаемой строкой. Как и для библиотеки `libpq`, объявим курсор, который будет применяться для прохода по совокупности возвращенных строк. Курсор будет действовать как закладка, и строки будут продолжать выбираться до тех пор, пока данные не закончатся.

Для работы с курсором следует объявить его и указать соответствующий ему запрос. Объявление курсора можно применять только внутри транзакции PostgreSQL, поэтому если не установлен ре-

жим автоматических транзакций ecpg, то необходимо начать транзакцию:

```
EXEC SQL begin work
EXEC SQL declare mycursor cursor for SELECT ... ;
```

Оператор SELECT, определяющий курсор, может содержать переменные основного языка в условиях инструкций WHERE и т. д., но в нем не должно быть инструкции INTO, т. к. на этом этапе извлечение данных в переменные основного языка не осуществляется.

Следующим шагом будет открытие курсора. Теперь он готов для выборки результатов:

```
EXEC SQL open mycursor;
```

Начинаем извлекать строки: выполняем FETCH с инструкцией INTO:

```
EXEC SQL fetch next from mycursor into :var1, :var2, ... ;
```

Когда строк для выборки не останется, счетчик строк в sqlca.sqlerrd[2] станет равен нулю, sqlca.sqlcode получит значение 100, и будет выполнено действие, указанное в коде sql whenever not found.

По окончании работы с курсором закрываем его и завершаем транзакцию:

```
EXEC SQL close mycursor;
EXEC SQL commit work;
```

Попробуйте сами. Курсоры

Рассмотрим программу (cursor.pgc), в которой курсор применяется для извлечения результатов запроса, подобного приведенному в главе 7: выбираются даты размещения заказов клиентами, живущими в определенном городе:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

EXEC SQL include sqlca;
EXEC SQL whenever sqlwarning sqlprint;
EXEC SQL whenever sqlerror do GiveUp();

void GiveUp()
{
    fprintf(stderr, "Fatal Error\n");
    sqlprint();
    exit(1);
}
```

```
main(int argc, char *argv[])
{
  EXEC SQL begin declare section;
  char town[32];
  double shipping;
  char date[15];
  EXEC SQL end declare section;

  if(argc != 2) {
    printf("Usage: cursor town\n");
    exit(1);
  }

  strncpy(town, argv[1], sizeof(town));

  EXEC SQL connect to bpsimple;
  EXEC SQL begin work;
  EXEC SQL declare mycursor cursor for
    SELECT oi.date_placed, oi.shipping
         FROM orderinfo oi, customer c
         WHERE oi.customer_id = c.customer_id
         AND town = :town;

  EXEC SQL open mycursor;

  EXEC SQL fetch next from mycursor into :date, :shipping;

  while(sqlca.sqlcode == 0) {
    printf("%.*s <%.2f>\n", sizeof(date), date, shipping);
    EXEC SQL fetch next from mycursor into :date, :shipping;
  }

  EXEC SQL close mycursor;
  EXEC SQL commit work;
  EXEC SQL disconnect all;

  return EXIT_SUCCESS;
}
```

Как это работает

Теперь программа аккуратно обрабатывает все три возможных случая, а именно отсутствие заказов, наличие ровно одного или же многих заказов:

```
$ make cursor
$ ./cursor Erewhon
$ ./cursor Milltown
2000-09-02 <3.99>
```

```
$ ./cursor Bingham
2000-06-23 <0.00>
2000-07-21 <5.55>
```

Отладка кода ecpg

Несмотря на то что ecpg хорошо генерирует C-код из файлов pgc, иногда при компиляции могут возникнуть проблемы. Обычно это объясняется ошибкой в коде C, а не тем, что ecpg сделала что-то не так. Номера строк следует искать в C-коде, сгенерированном ecpg.

Необходимо пойти на небольшую хитрость, удалив вставленные ecpg директивы препроцессора #line, в результате действия которых обычно ошибки компилятора ссылаются на исходный файл .pgc, а не на компилируемый на самом деле файл .c.

Опишем первые шаги:

- Вручную запустить ecpg для генерирования файла .c из файла .pgc
- С помощью grep удалить директивы #line
- Переместить временный файл обратно на его законное место
- Разрешить компиляции продолжиться или вручную вызвать компилятор C

Вот так, например, это можно сделать для cursor.pgc:

```
$ ecpg -t -I/usr/local/pgsql/include cursor.pgc
$ grep -v '^#line' cursor.c > _1.c
$ mv _1.c cursor.c
$ make
cc -I/usr/local/pgsql/include  -L/usr/local/pgsql/lib -lecpg -lpq  cursor.c
-o cursor
```

Резюме

В данной главе было рассказано о том, как использовать SQL в программах, написанных на C, встраивая операторы SQL непосредственно в исходные тексты. Транслятор ecpg генерирует C-код, обрабатываемый компилятором и преобразуемый в исполняемый код.

Охвачено соединение с базой данных и обработка возможных ошибок. Рассмотрено применение переменных основного языка для предоставления значений операторам INSERT и UPDATE.

Описана реализация простых операторов SELECT, извлечение строк данных в переменные основного языка и использование таких переменных для задания части инструкции WHERE. Показано, как распознавать неизвестные значения в извлекаемых данных при помощи индикаторных переменных. Наконец, для извлечения множества строк, возвращаемых более сложными запросами, был применен курсор.

Эта глава основывалась на сведениях, полученных в предыдущей, но рассказывала о более переносимом способе стыковки PostgreSQL и C. Можно сказать, что применение `libpq` обеспечивает чуть более полный контроль над результирующими множествами и информацией о состояниях, к тому же она поддерживает асинхронный режим работы. С другой стороны, встраивание SQL облегчает работу с двоичными значениями, т. к. `ecpg` заботится обо всем необходимом, встроенный SQL более переносим, и читать операторы SQL в программе гораздо легче. Выбор способа зависит от конкретных требований, предъявляемых к вашему приложению.

15

Доступ к PostgreSQL из PHP

В последнее время ощущается сильная тенденция к использованию веб-интерфейса к базам данных. Тому есть несколько причин, среди которых следующие:

- Веб-броузеры стали привычным и общедоступным средством доступа к данным
- Веб-ориентированные приложения легко встраиваются в уже существующий веб-сайт
- Веб-интерфейсы на основе HTML просты в создании и модификации

В этой главе рассматриваются различные способы доступа к PostgreSQL из PHP. PHP – это межплатформенный язык серверных сценариев, предназначенный для написания веб-ориентированных приложений. Он интерпретируется веб-сервером и позволяет встраивать в HTML-страницы программную логику, динамически формирующую содержимое. На PHP можно создавать веб-приложения, взаимодействующие с PostgreSQL.

В данной главе предполагается, что читатель знает как минимум основы PHP. Тем, кто вообще не знаком с этим языком, возможно, понадобятся следующие источники:

- Домашняя страница PHP: *http://www.php.net*
- «Основы PHP4» («Beginning PHP4»), Ванкиу Чой (Wankyu Choi), Алан Кент (Allan Kent), Ганеш Прасад (Ganesh Prasad) и Крис Уллман (Chris Ullman), при участии Джона Бланка (Jon Blank) и Шона Кэзела (Sean Cazzell), Wrox Press (ISBN 1-861003-73-0)

Существует несколько различных подходов к методологии программирования на PHP. Их обсуждение не входит в круг вопросов, рассматриваемых в данной книге. Мы сосредоточимся на вопросах разработки сценариев на PHP, эффективно использующих имеющийся интерфейс с PostgreSQL.

Обратите внимание, что будет рассматриваться PHP версии 4. Несмотря на то что большинство примеров и пояснений будут справедливы и для более ранних версий PHP, возможны и некоторые отличия в функциональности. Кроме того, предполагается, что все фрагменты кода находятся в области видимости PHP (обычно ограниченной тегами `<?php ?>`), если не оговорено обратное.

Установка поддержки PostgreSQL в PHP

Прежде чем приступать к написанию PHP-сценариев, взаимодействующих с базой данных PostgreSQL, необходимо установить PHP с поддержкой PostgreSQL.

Если вы затрудняетесь определить, включена ли поддержка PostgreSQL в установленном у вас PHP, создайте простой сценарий с именем `phpinfo.php` (который следует поместить в корневой каталог документов веб-сервера), содержащий следующие строки:

```
<?php
phpinfo();
?>
```

Просмотрите результат выполнения этого сценария в броузере. Если поддержка PostgreSQL уже была включена, то должен появиться раздел приблизительно такого вида:

pgsql

PostgreSQL	**Enabled**
Active Persistent Links	0
Active Links	0

Directive	Local Value	Master Value
Pgsql.allow_persistent	On	On
Pgsql.max_links	Unlimited	Unlimited
Pgsql.max_persistent	Unlimited	Unlimited

Если PHP уже установлен с поддержкой PostgreSQL, можете переходить к следующему разделу.

При наличии исходного кода PHP добавление поддержки PostgreSQL не составит труда. Просто укажите параметр `--with-pgsql` при запуске сценария конфигурации:

```
$ ./configure --with-pgsql
```

Если сценарий затрудняется сам определить, в каком каталоге установлен PostgreSQL, можно указать этот путь в параметре:

```
$ ./configure --with-pgsql=/var/lib/pgsql
```

Помните, что вам, возможно, потребуется передать сценарию `configure` и другие параметры для получения требуемой конфигурации. Например, для сборки PHP с поддержкой PostgreSQL, LDAP и XML командная строка должна иметь такой вид:

```
$ ./configure --with-pgsql --with-ldap --enable-xml
```

Описание дополнительных параметров компиляции и инструкции по установке можно найти в документации по PHP (в частности, в документе `INSTALL`, входящем в дистрибутив) и на странице *http://www.php.net/manual/en/html/installation.html*.

Использование PHP API для PostgreSQL

Все взаимодействие с базой данных PostgreSQL осуществляется посредством расширения PostgreSQL, представляющего собой обширный набор функций PHP. Полный перечень функций и дополнительную информацию о них можно получить по адресу *http://www.php.net/manual/ref.pgsql.php*.

Простой сценарий на PHP, устанавливающий соединение с PostgreSQL, выбирающий несколько строк, выводящий количество полученных строк и закрывающий соединение, будет выглядеть примерно так:

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
$query = "SELECT * FROM item";
$result = pg_exec($db_handle, $query);
echo "Number of rows: " . pg_numrows($result);
pg_freeresult($result);
pg_close($db_handle);
?>
```

Как видите, работать с базой данных из PHP достаточно просто. Теперь рассмотрим более подробно различные аспекты использования расширения PostgreSQL для PHP.

Соединения с базой данных

Прежде чем начать работу с базой данных, необходимо установить с ней соединение. Каждое соединение представляется отдельной переменной (которую мы будем называть **дескриптором соединения**). PHP позволяет открыть несколько соединений одновременно, при этом каждое из них имеет собственный дескриптор.

pg_connect()

Соединение с базой данных устанавливается с помощью функции `pg_connect()`. Эта функция принимает в качестве единственного аргумента строку соединения и возвращает дескриптор соединения с базой данных. Вот пример:

```
$db_handle = pg_connect("dbname=bpsimple user=jon");
```

Можете создать собственного пользователя и указать его имя для соединения в виде `user=<username>`.

Обращаясь к переменным PHP, не забывайте брать строку соединения в двойные кавычки, а не в одинарные:

```
$db_handle = pg_connect("dbname=$dbname user=$dbuser");
```

В строке соединения можно указывать все стандартные параметры соединения PostgreSQL. Наиболее употребительные параметры и их значения приведены в табл. 15.1:

Таблица 15.1. Параметры соединений

Параметр	Значение	По умолчанию
`Dbname`	База данных, с которой устанавливается соединение	`$PGDATABASE`
`User`	Имя пользователя, устанавливающего соединение	`$PGUSER`
`password`	Пароль указанного пользователя	`$PGPASSWORD` или не указан
`Host`	Имя сервера, с которым устанавливается соединение	`$PGHOST` или `localhost`
`hostaddr`	IP-адрес сервера, с которым устанавливается соединение	`$PGHOSTADDR`
`Port`	Порт TCP/IP, с которым устанавливается соединение	`$PGPORT` or 5432

Если попытка соединиться оказалась неуспешной, функция `pg_connect()` вернет значение «ложь». Таким образом, проверяя возвращаемое значение, можно определить, было ли установлено соединение:

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
if ($db_handle) {
```

```
    echo 'Connection attempt succeeded.';
} else {
    echo 'Connection attempt failed.';
}
pg_close($db_handle);
?>
```

Как уже говорилось, PHP поддерживает несколько одновременных соединений с базой данных:

```
$db_handle1 = pg_connect("dbname=database1");
$db_handle2 = pg_connect("dbname=database2");
```

Постоянные соединения

PHP поддерживает также **постоянные** соединения с базой данных. Такие соединения остаются открытыми после того, как обращение к странице закончено, в отличие от обычных соединений, которые закрываются после обращения к странице. PHP хранит список открытых в данный момент соединений и, получив запрос на установление нового постоянного соединения с параметрами, соответствующими какому-либо соединению из списка, возвращает уже имеющийся дескриптор. Преимущество этого способа в том, что он позволяет избежать издержек на создание нового соединения с базой данных, если подходящее уже имеется в пуле соединений.

pg_pconnect()

Для открытия постоянного соединения с PostgreSQL предназначена функция `pg_pconnect()`. Она работает аналогично описанной раньше `pg_connect()` за исключением того, что при наличии постоянного соединения использует его.

Имея дело с постоянными соединениями, следует соблюдать некоторую осторожность. В результате злоупотребления ими может образоваться большое количество неиспользуемых соединений с базой данных. Постоянные соединения идеально подходят для случаев, когда несколько страниц требуют одного и того же типа соединения с базой данных (то есть с теми же самыми параметрами соединения). В таких ситуациях постоянные соединения реально увеличивают производительность.

Закрытие соединений

pg_close()

Соединения с базой данных должны быть явно закрыты функцией `pg_close()`:

```
pg_close($db_handle);
```

Однако необходимо сделать несколько замечаний. Во-первых, эта функция на самом деле не закрывает соединение. Вместо этого соединение возвращается в пул соединений с базой данных. Во-вторых, PHP автоматически закрывает все открытые непостоянные соединения с базой данных в конце выполнения сценария. На основе этих двух пунктов можно прийти к выводу, что вызов функции `pg_close()` не так уж и необходим, но все же упомянем о ней для полноты описания, а также потому, что в некоторых случаях действительно требуется незамедлительно закрыть соединение.

Если представленный дескриптор соединения недействителен, то `pg_close()` вернет значение «ложь», в остальных случаях функция возвращает «истину».

Информация о соединении

PHP предусматривает ряд простых функций для получения информации о текущем соединении с базой данных по его дескриптору. Перечень таких функций приведен в табл. 15.2:

Таблица 15.2. Функции, предоставляющие сведения о соединении

Функция	Описание
`pg_dbname()`	Возвращает имя текущей базы данных
`pg_host()`	Возвращает имя сервера, соответствующего текущему соединению
`pg_options()`	Возвращает параметры текущего соединения
`pg_port()`	Возвращает порт текущего соединения
`pg_tty()`	Возвращает TTY-имя для текущего соединения

Единственным аргументом всех этих функций является дескриптор соединения, а возвращают они (в случае успеха) или строку, или число. В случае же неудачи функции возвращают «ложь»:

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
echo "<h1>Connection Information</h1>";
echo "Database name: " . pg_dbname($db_handle) . "<br>\n";
echo "Hostname: " . pg_host($db_handle) . "<br>\n";
echo "Options: " . pg_options($db_handle) . "<br>\n";
echo "Port: " . pg_port($db_handle) . "<br>\n";
echo "TTY name: " . pg_tty($db_handle) . "<br>\n";
pg_close($db_handle);
?>
```

Построение запросов

Простой пример выполнения запроса из PHP уже был рассмотрен. В данном разделе мы углубимся в построение и выполнение запросов.

Запросы SQL – это просто строки, поэтому запросы могут быть построены при помощи любых строковых функций PHP. Далее приведены три примера составления строки запроса в PHP:

```
$lastname = strtolower($lastname);
$query = "SELECT * FROM customer WHERE lname = '$lastname'";
```

Этот пример сначала производит преобразование `$lastname` к нижнему регистру, а затем строит строку запроса, используя стандартные операции со строками PHP.

Отметьте, что после этих строк значение `$lastname` останется в нижнем регистре.

```
$query = "SELECT * FROM customer WHERE lname = '" .strtolower($lastname) . "'";
```

В данном примере внутри строки выполняется вызов `strtolower()`. Функции не могут вызываться из строковых констант (то есть слов, заключенных в кавычки), поэтому следует разбить строку запроса на две части и соединить их (применив оператор конкатенации «точка»), а функцию вызвать между ними.

В отличие от предыдущего примера, результат функции `strtolower()` не влияет на значение `$lastname` после того, как строка выполняется PHP.

```
$query = sprintf("SELECT * FROM customer WHERE lname = '%s'",
strtolower($lastname));
```

В последнем примере для генерирования строки запроса используется функция `sprintf()`. Функция `sprintf()` применяет комбинации специальных символов (например, `%s` в строке, приведенной выше) для форматирования строк. Подробную информацию о функции `sprintf()` можно получить по адресу *http://www.php.net/manual/en/function.sprintf.php*.

Каждый из этих способов порождает одну и ту же строку запроса. Какой из них следует предпочесть, как и для многих других вещей, зависит от ситуации. Для простых запросов, вероятно, лучше прямое присваивание, если же требуется вставка или преобразование большого количества переменных, стоит проанализировать различные методы. В некоторых случаях необходимо пойти на компромисс для достижения лучшего соотношения «скорость выполнения – читаемость кода». Это справедливо для большинства задач программирования, и выбор здесь остается за вами.

Приведем пример сложного запроса, записанного в виде длинной строки присваивания:

```
$query = "UPDATE table $tablename SET " . strtolower($column) . " = '" .
strtoupper($value) . "'";
```

Можно переписать его при помощи функции PHP `sprintf()` так:

```
$query = sprintf("UPDATE table %s SET %s = '%s'", $tablename,
strtolower($column), strtoupper($value));
```

Второе выражение очевидно более прозрачно, чем первое, хотя измерение производительности показало бы, что легкость чтения достигается за счет небольшой потери скорости работы, но время программиста стоит гораздо дороже машинного. В данном случае, наверное, удобочитаемость важнее, чем скорость выполнения, если только вы не пишете такие строки сотнями на каждой странице.

Сложные запросы

В идеальном мире все запросы были бы столь же простыми, как приведенные в предыдущих примерах, в действительности же так бывает нечасто. Для построения более сложных запросов PHP предоставляет ряд удобных функций, которые помогают решить задачу.

Рассмотрим, например, случай, когда нужно произвести большое количество удалений в таблицах. В чистом SQL запрос мог бы выглядеть примерно так:

```
DELETE FROM items WHERE item_id = 4 OR item_id = 6
```

Сейчас запрос не кажется сложным, но что если потребуется удалить дюжину строк, указывая `item_id` каждой в инструкции `WHERE`? Строка запроса становится очень длинной, к тому же количество выражений в инструкции `where` могло бы меняться, в связи с чем необходимо принять это во внимание в нашем коде.

Вероятно, список идентификаторов товаров, подлежащих удалению, будет поступать от пользователя из какой-то формы HTML, поэтому можно предположить, что они будут сохранены в каком-то стандартном формате (по крайней мере, такой способ хранения списка наиболее удобен). Пусть массив идентификаторов товаров называется `$item_ids`. Основываясь на этом предположении, построим упомянутый выше запрос следующим образом:

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
$query = "DELETE FROM items WHERE ";
$query .= "item_id = " . $item_ids[0];
if (count($item_ids) > 1) {
array_shift($item_ids);
$query .= " or item_id = " .
implode(" or item_id =", $item_ids);
pg_close($db_handle);
?>
```

Будет сгенерирован запрос SQL для произвольного количества идентификаторов товаров. На базе такого кода можно написать универсальную функцию, выполняющую необходимые удаления:

```
<?php
$db_handle = pg_connect("dbname=bpsimple");

function sqlDelete($tablename, $column, $ids)
{
    $query = '';
    if (is_array($ids)) {
        $query = "DELETE FROM $tablename WHERE ";
        $query .= "$column = " . $ids[0];
        if (count($ids) > 1) {
            array_shift($ids);
            $query .= " or $column = " .
                implode(" or $column =", $ids);
        }
    }
    return $query;
}
pg_close($db_handle);
?>
```

Выполнение запросов

Построив строку запроса, можно приступать к ее выполнению. Запросы выполняются при помощи функции `pg_exec()`.

pg_exec()

Функция `pg_exec()` отвечает за отправку строки запроса на сервер PostgreSQL и возврат результирующего множества.

Вот простой пример, иллюстрирующий применение `pg_exec()`:

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
$query = 'SELECT * FROM customer';
$result = pg_exec($db_handle, $query);
pg_close($db_handle);
?>
```

Как видите, `pg_exec()` нуждается в двух параметрах: дескрипторе активного соединения и строке запроса (оба были рассмотрены в предыдущих разделах). В случае успешного выполнения запроса `pg_exec()` возвращает результирующее множество. О результирующих множествах речь пойдет в следующем разделе.

Если не удается выполнить запрос или же указан недействительный дескриптор соединения, то `pg_exec()` возвращает значение «ложь».

Поэтому разумно проверять значение, возвращенное pg_exec(), чтобы обнаружить подобные ошибки.

Следующий пример содержит проверку результата:

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
$query = "SELECT * FROM customer";
$result = pg_exec($db_handle, $query);
if ($result) {
    echo "The query executed successfully.<br>\n";
} else {
    echo "The query failed with the following error:<br>\n";
    echo pg_errormessage($db_handle);
}
pg_close($db_handle);
?>
```

В этом примере проверяется значение, возвращенное pg_exec(). Если оно не равно FALSE (другими словами, имеет значение), то $result представляет собой результирующее множество. В противном случае ($result имеет значение «ложь») ясно, что произошла ошибка. Функция pg_errormessage() позволяет вывести сообщения с описанием данной ошибки. Сообщения об ошибках более подробно обсуждаются чуть позже.

Работа с результирующими множествами

В случае успешного выполнения запроса pg_exec() возвращает идентификатор результирующего множества, по которому можно получить доступ к результату. Результирующее множество хранит результат запроса в том виде, в каком он возвращен базой данных. Например, если выполнялась выборка, то результирующее множество будет содержать строки таблицы.

PHP предоставляет несколько функций для работы с результирующими множествами. Все они принимают в качестве аргумента идентификатор результирующего множества, поэтому вызывать их можно лишь после успешного выполнения запроса. Проверять, успешно ли выполнен запрос, мы научились в предыдущем разделе.

pg_numrows() и pg_numfields()

Начнем с двух самых простых функций результата: pg_numrows() и pg_numfields(). Они возвращают количество строк и полей в результирующем множестве соответственно. Например:

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
$query = "SELECT * FROM customer";
$result = pg_exec($db_handle, $query);
```

```
if ($result) {
    echo "The query executed successfully.<br>\n";
    echo "Number of rows in result: " . pg_numrows($result) . "<br>\n";
    echo "Number of fields in result: " . pg_numfields($result);
} else {
    echo "The query failed with the following error:<br>\n";
    echo pg_errormessage($db_handle);
}
pg_close($db_handle);
?>
```

Если возникает ошибка, то функции возвращают `-1`.

pg_cmdtuples()

Есть еще функция `pg_cmdtuples()`, которая возвращает количество строк, **затронутых** запросом. Например, если в запросе выполняются вставки или удаления, то фактически из базы данных не будет извлечено ни одной строки, поэтому количество строк или полей результирующего множества не будет характеризовать результат. Но внутри базы данных происходят изменения, и функция `pg_cmdtuples()` возвращает количество строк, затронутых таким видом запросов (то есть количество вставленных, удаленных или обновленных строк):

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
$query = "DELETE FROM item WHERE cost_price > 10.00";
$result = pg_exec($db_handle, $query);
if ($result) {
    echo "The query executed successfully.<br>\n";
echo "Number of rows deleted: " . pg_cmdtuples($result);
} else {
    echo "The query failed with the following error:<br>\n";
    echo pg_errormessage($db_handle);
}
pg_close($db_handle);
?>
```

Если запрос не изменил ни одной строки базы данных (осуществлялась выборка), то `pg_cmdtuples()` вернет `0`.

Извлечение значений из результирующих множеств

Существует несколько способов извлечения значений из результирующего множества. Начнем с рассмотрения функции `pg_result()`.

pg_result()

Функция `pg_result()` применяется, если необходимо извлечь одно отдельное значение из результирующего множества. Вдобавок к идентификатору множества можно указать строку и поле, которое требуется извлечь из результата. Строки задаются числами, а поле может быть

задано как по имени, так и с помощью числового индекса. Нумерация всегда начинается с нуля.

Приведем пример использования `pg_result()`:

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
$query = "SELECT title, fname, lname FROM customer";
$result = pg_exec($db_handle, $query);
if ($result) {
    echo "The query executed successfully.<br>\n";
for ($row = 0; $row < pg_numrows($result); $row++) {
    $fullname = pg_result($result, $row, 'title') . " ";
    $fullname .= pg_result($result, $row, 'fname') . " ";
    $fullname .= pg_result($result, $row, 'lname');
    echo "Customer: $fullname<br>\n";
    }
} else {
    echo "The query failed with the following error:<br>\n";
    echo pg_errormessage($db_handle);
}
pg_close($db_handle);
?>
```

С помощью числовых индексов тот же самый блок кода можно записать следующим образом:

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
$query = "SELECT title, fname, lname FROM customer";
$result = pg_exec($db_handle, $query);
if ($result) {
    echo "The query executed successfully.<br>\n";
for ($row = 0; $row < pg_numrows($result); $row++) {
    for ($col = 0; $col < pg_numfields($result); $col++) {
    $fullname = pg_result($result, $row, $col) . " ";
        $fullname .= pg_result($result, $row, $col) . " ";
        $fullname .= pg_result($result, $row, $col);
        echo "Customer: $fullname<br>\n";
        }
    }
} else {
    echo "The query failed with the following error:<br>\n";
    echo pg_errormessage($db_handle);
}
pg_close($db_handle);
?>
```

Первый пример несколько более удобочитаем и не зависит от порядка полей в результирующем множестве. PHP предоставляет и более продвинутые способы извлечения данных, т. к. перемещение по строкам результата не слишком эффективно.

pg_fetch_row()

В PHP есть две функции, pg_fetch_row() и pg_fetch_array(), способные возвращать сразу несколько значений результата. Каждая из этих функций возвращает массив.

Функция pg_fetch_row()возвращает массив, соответствующий одной строке результирующего множества. Нумерация элементов массива начинается с нуля. Перепишем предыдущий пример, используя pg_fetch_row():

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
$query = "SELECT title, fname, lname FROM customer";
$result = pg_exec($db_handle, $query);
if ($result) {
    echo "The query executed successfully.<br>\n";
for ($row = 0; $row < pg_numrows($result); $row++) {
    $values = pg_fetch_row($result, $row);
    for ($col = 0; $col < count($values); $col++) {
    $fullname = $values[$col] . " ";
        $fullname .= $values[$col] . " ";
        $fullname .= $values[$col];
    echo "Customer: $fullname<br>\n";
        }
    }
} else {
    echo "The query failed with the following error:<br>\n";
    echo pg_errormessage($db_handle);
}
pg_close($db_handle);
?>
```

Как видите, применение pg_fetch_row() устраняет многочисленные вызовы функции pg_result(). К тому же значения помещаются в массив, который можно без труда обработать, используя собственные функции PHP для работы с массивами.

Однако в этом примере доступ к полям все еще осуществляется по их числовым индексам. В идеале хотелось бы получать доступ к каждому полю по его имени. Добиться этого можно, обратившись к функции pg_fetch_array().

pg_fetch_array()

Функция pg_fetch_array() также возвращает массив, но она позволяет указать способ индексации массива – численным или же ассоциативным индексом (с именами полей в качестве ключей). Способ индексации указывается как третий аргумент функции, возможные варианты представлены в табл. 15.3:

Таблица 15.3. Способы индексирования массивов значений результата

Вариант	Описание
PGSQL_ASSOC	Индексировать массив по имени поля
PGSQL_NUM	Индексировать массив численно
PGSQL_BOTH	Индексировать массив как численно, так и по имени поля

Если способ индексирования не указан, то по умолчанию выбирается PGSQL_BOTH. Обратите внимание, что это удвоит размер результирующего множества, поэтому, вероятно, лучше явно определить один из вышеперечисленных способов. Отметьте также, что названия полей всегда возвращаются в нижнем регистре, вне зависимости от того, как они представлены в базе данных.

Еще раз перепишем наш пример, на этот раз применяя функцию pg_fetch_array():

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
$query = "SELECT title, fname, lname FROM customer";
$result = pg_exec($db_handle, $query);
if ($result) {
    echo "The query executed successfully.<br>\n";
for ($row = 0; $row < pg_numrows($result); $row++) {
    $values = pg_fetch_array($result, $row, PGSQL_ASSOC);
$fullname = $values['title'] . " ";
    $fullname .= $values['fname'] . " ";
    $fullname .= $values['lname'];
echo "Customer: $fullname<br>\n";
    }
} else {
    echo "The query failed with the following error:<br>\n";
    echo pg_errormessage($db_handle);
}
pg_close($db_handle);
?>
```

pg_fetch_object()

PHP позволяет выбирать результирующие значения и при помощи функции pg_fetch_object(). Каждое имя поля будет представлено как свойство данного объекта. Поэтому к полю нельзя получить доступ с помощью числового индекса. Написанный с использованием pg_fetch_object() пример выглядит следующим образом:

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
$query = "SELECT title, fname, lname FROM customer";
$result = pg_exec($db_handle, $query);
if ($result) {
```

```
    echo "The query executed successfully.<br>\n";
for ($row = 0; $row < pg_numrows($result); $row++) {
    $values = pg_fetch_object($result, $row, PGSQL_ASSOC);
$fullname = $values->title . " ";
    $fullname .= $values->fname . " ";
    $fullname .= $values->lname;
echo "Customer: $fullname<br>\n";
    }
} else {
    echo "The query failed with the following error:<br>\n";
    echo pg_errormessage($db_handle);
}
pg_close($db_handle);
?>
```

Информация о полях

PHP позволяет собрать некоторую информацию о значениях полей результирующего множества. При определенных обстоятельствах такая возможность может быть полезной, поэтому кратко представим обеспечивающие ее функции.

pg_fieldisnull()

PostgreSQL поддерживает понятие неопределенного значения поля (`NULL`). Но PHP не обязательно определяет `NULL` так же, как PostgreSQL. Поэтому в PHP есть функция `pg_fieldisnull()`, позволяющая понять, является ли значение поля равным `NULL` в терминах PostgreSQL:

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
$query = "SELECT title, fname, lname FROM customer";
$result = pg_exec($db_handle, $query);
if (pg_fieldisnull($result, $row, $field)) {
    echo "$field is NULL.";
} else {
    echo "$field is " . pg_result($result, $row, $field);
}
pg_close($db_handle);
?>
```

pg_fieldname() и pg_fieldnum()

Эти функции возвращают имя или номер заданного поля. Поля проиндексированы числами начиная с нуля:

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
$query = "SELECT title, fname, lname FROM customer";
$result = pg_exec($db_handle, $query);
if (pg_fieldisnull($result, $row, $field)) {
```

```
    echo "$field is NULL.";
    echo "Field 1 is named: " . pg_fieldname($result, 1);
    echo "Field item_id is number: " . pg_fieldnum($result, "item_id");
} else {
    echo "$field is " . pg_result($result, $row, $field);
}
pg_close($db_handle);
?>
```

Обратите внимание, что `pg_fieldname()` вернет то имя поля, которое указано в операторе `SELECT`.

pg_fieldsize(), pg_fieldprtlen() и pg_fieldtype()

Можно определить размер, количество печатаемых символов и тип полей:

```
<?php
$db_handle = pg_connect("dbname=bpsimple");
$query = "SELECT title, fname, lname FROM customer";
$result = pg_exec($db_handle, $query);
if (pg_fieldisnull($result, $row, $field)) {
    echo "$field is NULL.";
    echo "Field 1 is named: " . pg_fieldname($result, 1);
    echo "Field 1 is named: " . pg_fieldname($result, 1);
    echo "Field item_id is number: " . pg_fieldnum($result, "item_id");
    echo  "Size of field 2:" . pg_fieldsize($result, 2);
    echo "Length of field 2: " . pg_fieldprtlen($result, $row, 2);
    echo "Type of field 2: " . pg_fieldtype($result, 2);
} else {
    echo "$field is " . pg_result($result, $row, $field);
}
pg_close($db_handle);
?>
```

Как обычно, числовые индексы полей начинаются с 0. Поля могут быть проиндексированы и строками, представляющими собой имена полей.

Если поле имеет переменный размер, то `pg_fieldsize()` возвращает `-1`, а если возникает ошибка, то «ложь». Функция же `pg_fieldprtlen()` возвращает `-1` в случае ошибки.

Освобождение результирующих множеств

pg_freeresult()

Посредством функции `pg_freeresult()` можно освободить память, занятую результирующим множеством:

```
pg_freeresult($result);
```

PHP автоматически высвобождает всю память, отведенную под результаты, в конце выполнения сценария, поэтому данную функцию стоит вызывать, только если вы очень обеспокоены потреблением памяти в сценарии и уверены, что это результирующее множество не понадобится для дальнейшего выполнения сценария.

Преобразование типов для значений результата

В отличие от других языков, PHP не предоставляет поддержку разнообразных типов данных, поэтому значения результирующих множеств иногда преобразуются из их исходного типа данных в собственный тип PHP. По большей части такие преобразования очень мало влияют на ваше приложение (или даже совсем не влияют), но важно знать, что они происходят:

- Все целые, логические и OID-типы (идентификаторы объектов) преобразуются в целый тип
- Все формы чисел с плавающей точкой преобразуются в тип `double`
- Все остальные типы (массивы и т. д.) представляются в виде строк

Обработка ошибок

Об обработке ошибок ранее лишь упоминалось, теперь же поговорим о ней более подробно.

Практически все относящиеся к PostgreSQL функции возвращают какое-то предсказуемое значение в случае ошибки (обычно `false` или `-1`). Благодаря этому обстоятельству обнаружить ошибки достаточно просто, так что и в этом случае ваш сценарий сможет завершиться вполне элегантно. Например:

```
$db_handle = pg_connect('dbname=bpsimple');
if (!$db_handle) {
    header("Location: http://www.example.com/error.php");
    exit;
}
```

В примере, приведенном выше, пользователь перенаправляется на страницу ошибок, если не удалось установить соединение с базой данных.

pg_errormessage()

Функция `pg_errormessage()` применяется для извлечения текста сообщения об ошибке в том виде, в котором он возвращен сервером базы данных. Всегда возвращается текст последнего сообщения об ошибке, сгенерированного сервером, не забудьте об этом, проектируя логику обработки и отображения ошибок.

В зависимости от выбранного уровня сообщений об ошибках PHP может быть весьма многословен, часто выводя по несколько строк ошибок и предупреждений. В рабочей среде вывод таких сообщений для конечного пользователя может быть нежелательным.

Символ @

Прямое решение состоит в понижении уровня сообщений об ошибках в PHP (он управляется посредством переменной конфигурации `error_reporting` из `php.ini`). В качестве альтернативы можно предложить подавлять сообщения об ошибках непосредственно в коде PHP при вызове каждой функции. В PHP для указания на то, что ошибка должна быть скрыта, применяется символ `@`. Так, приведенный ниже код не вызовет никаких сообщений об ошибках:

```
$db_handle = pg_connect("host=nonexistent_host");
$result = @pg_exec($db_handle, "SELECT * FROM item");
```

Если бы во второй строке не было символа `@`, то она бы сгенерировала сообщение об ошибке, уведомляющее об отсутствии установленного соединения с базой данных (если, конечно, установленный уровень сообщений об ошибках достаточно высок для того, чтобы вызвать появление такого сообщения).

Однако такую ошибку все еще можно выявить, проверив значение `$result`, поэтому подавление вывода сообщений об ошибках не мешает программной обработке ошибочных ситуаций. Более того, можно вывести сообщение об ошибке по своему усмотрению, вызвав функцию `pg_errormessage()`.

Если (`!result ...`) выводит ошибку, значит, SQL-запрос содержит ошибку. Если (`!pg_cmdTuples($result)`) выводит ошибку, значит, по крайней мере одно из значений в SQL-запросе содержит ошибку.

Таблицы кодировки

Если поддержка кодовых таблиц включена в PostgreSQL, то PHP предоставляет функции для получения и установки текущей клиентской кодировки. По умолчанию применяется кодировка SQL ASCII.

Поддерживаются следующие наборы символов: `SQL_ASCII`, `EUC_JP`, `EUC_CN`, `EUC_KR`, `EUC_TW`, `UNICODE`, `MULE_INTERNAL`, `LATINX (X=1...9)`, `KOI8`, `WIN`, `ALT`, `SJIS`, `BIG5`, `WIN1250`.

pg_client_encoding()

Функция `pg_client_encoding()` возвращает текущую клиентскую кодировку:

```
$encoding = pg_client_encoding($db_handle);
```

pg_set_client_encoding()

Установить текущую клиентскую кодировку можно при помощи функции `pg_set_client_encoding()`:

```
pg_set_client_encoding($db_handle, 'UNICODE');
```

PEAR

Проект PEAR (The PHP Extension and Application Repository – репозитарий расширений и программ языка PHP) представляет собой попытку повторить идею CPAN языка Perl для PHP. Основные цели PEAR:

- Предоставить библиотечную инфраструктуру, в которую авторы могли бы выкладывать свои коды вместе с другими разработчиками, открыв их для общего доступа
- Создать инфраструктуру для коллективно используемого кода для сообщества PHP
- Определить стандарты, призванные помочь разработчикам писать переносимый и допускающий многократное использование код
- Предоставить средства для поддержки и распространения кода

PEAR прежде всего является обширной коллекцией классов PHP, в которой реализованы возможности объектно-ориентированного программирования на PHP. Поэтому вам необходимо освоить синтаксис, принятый в PHP для работы с классами PHP. Документация по объектно-ориентированным расширениям PHP представлена по адресу *http://www.php.net/manual/en/language.oop.php*.

Дополнительная информация о PEAR доступна на веб-сайтах:

- *http://pear.php.net*
- *http://php.weblogs.com/php_pear_tutorials/*

Абстрактный интерфейс базы данных в PEAR

PEAR содержит абстрактный интерфейс базы данных (DB), который включен в стандартный дистрибутив PHP. Преимущество использования абстрактного DB-интерфейса вместо непосредственного вызова собственных функций базы данных заключается в независимости кода. Для переноса проекта на другую базу данных наверняка потребуется значительное изменение кода. Если же применялся абстрактный интерфейс, то задача будет почти тривиальной.

DB-интерфейс PEAR также вводит несколько дополнительных возможностей, таких как удобный доступ к нескольким результирующим множествам и интегрированная обработка ошибок. Все взаимодействие с базой данных управляется через классы и объекты DB. Идеологически это напоминает интерфейс Perl DBI.

Основной недостаток абстрактного интерфейса базы данных состоит в связанном с его применением увеличении вычислительной нагрузки.

Здесь опять необходим компромисс между гибкостью кода и производительностью.

Использование DB-интерфейса

Следующий пример иллюстрирует использование интерфейса базы данных: учтите, что здесь предполагается, что PEAR DB-интерфейс уже установлен и может быть найден по текущему значению `include_path`. В новых дистрибутивах PHP4 и то и другое верно по умолчанию:

```
<?php

/* импортировать PEAR-интерфейс БД. */
require_once "DB.php";

/* Параметры соединения с базой данных. */
$username = "jon";
$password = "secret";
$hostname = "localhost";
$dbname = "bpsimple";

/* Создать DSN - имя источника данных. */
$dsn = "pgsql://$username:$password@$hostname/$dbname";

/* Попытка соединиться с базой данных. */
$db = DB::connect($dsn);

/* Проверка на ошибки соединения. */
if (DB::isError($db)) {
    die ($db->getMessage());
}

/* Выполнение запроса-выборки. */
$query = "SELECT title, fname, lname FROM customer";
$result = $db->query($query);

/* Проверка на ошибки выполнения запроса. */
if (DB::isError($result)) {
    die ($result->getMessage());
}

/* Выбрать и вывести результаты запроса. */
while ($row = $result->fetchRow(DB_FETCHMODE_ASSOC)) {
$fullname = $row['title'] . " ";
    $fullname .= $row['fname'] . " ";
    $fullname .= $row['lname'];
echo "Customer: $fullname<br>\n";
}
```

```
/* Отключение от базы данных. */
$db->disconnect();

?>
```

Как видите, этот код хотя и не использует непосредственно функции PostgreSQL, но следует той же программной логике, что и предыдущие примеры. Несложно понять, как без труда адаптировать данный пример к другому типу базы данных (например, Oracle или MySQL).

Обработка ошибок в PEAR

Применение PEAR DB-интерфеса предоставляет разработчикам ряд дополнительных преимуществ. Например, PEAR включает в себя интегрированную систему обработки ошибок. Ниже приведен код, демонстрирующий обработку ошибок:

```
<?php

/* Импортировать PEAR-интерфейс БД. */
require_once 'DB.php';

/* Создать DSN - имя источника данных */
$dsn = "pgsql://jon:secret@localhost/bpsimple";

/* Попытка соединиться с базой данных. */
$db = DB::connect($dsn);

/* Проверка на ошибки соединения. */
if (DB::isError($db)) {
    die ($db->getMessage());
}
```

Выше представлен первый вариант обработки ошибок PEAR: `DB::isError()`. Если по какой-то причине вызов `DB::connect()` окончится неудачей, то возвращается экземпляр `PEAR_Error`, а не объект соединения с базой данных. Осуществить проверку можно при помощи функции `DB::isError()`, что и было сделано.

Знать, что произошла ошибка, очень важно, но еще важнее понять, почему так случилось. Можно извлечь текст сообщения об ошибке (в данном случае это ошибка соединения, сгенерированная PostgreSQL), обратившись к методу `getMessage()` объекта `PEAR_Error`. Это также было показано в примере.

Пример продолжается запросами:

```
/* Сделать ошибки фатальными. */
$db->setErrorHandling(PEAR_ERROR_DIE);

/* Построить и выполнить запрос. */
$query = "SELECT title, fname, lname FROM customer";
```

```
$result = $db->query($query);

/* Проверка на ошибки выполнения запроса. */
if (DB::isError($result)) {
    die ($result->getMessage());
}

while ($row = $result->fetchRow(DB_FETCHMODE_ASSOC)) {
$fullname = $row['title'] . " ";
    $fullname .= $row['fname'] . " ";
    $fullname .= $row['lname'];
echo "Customer: $fullname<br>\n";
}

/* Отключение от базы данных. */
$db->disconnect();

?>
```

Обратите внимание, что поведение PEAR при обработке ошибок было изменено вызовом метода `setErrorHandling()`. Установка режима обработки ошибок в значение `PEAR_ERROR_DIE` приведет к тому, что в случае ошибки выполнение PHP будет аварийно завершено.

Список других вариантов поведения обработчика ошибок приведен в табл. 15.4:

Таблица 15.4. Режимы обработки ошибок

Вариант	Описание
`PEAR_ERROR_RETURN`	Просто возвратить ошибочный объект (по умолчанию)
`PEAR_ERROR_PRINT`	Вывести сообщение об ошибке и продолжить выполнение
`PEAR_ERROR_TRIGGER`	Использовать PHP-функцию `trigger_error()` для порождения внутренней ошибки
`PEAR_ERROR_DIE`	Вывести сообщение об ошибке и прервать выполнение
`PEAR_ERROR_CALLBACK`	Использовать функцию обратного вызова для обработки ошибки перед прекращением выполнения

Дополнительную информацию о классе `PEAR_Error` и обработке ошибок в PEAR можно получить по адресу *http://php.net/manual/ class.pear-error.php*.

Подготовка и выполнение запроса

PEAR также предоставляет способ подготовки и выполнения запросов. Рассмотрим несколько сокращенный пример, демонстрирующий методы `prepare()` и `execute()` DB-интерфейса. В нем предполагается, что соединение с базой данных уже успешно установлено (с помощью `DB::connect()`):

```
/* Сформировать массив $items. */
$items = array(
```

```
    '6241527836190' => 20,
    '7241427238373' => 21,
    '7093454306788' => 22
);

/* Подготовить наш шаблонный оператор SQL. */
$statement = $db->prepare("INSERT INTO barcode VALUES(?,?)");

/* Выполнить оператор для каждого элемента массива $items. */
while (list($barcode, $item_id) = each($items)) {
    $db->execute($statement, array($barcode, $item_id));
}
```

Тем, кто не знаком с подготовленными операторами SQL, вероятно, требуются пояснения.

Вызов метода `prepare()` создает шаблон SQL, который может выполняться много раз. Обратите внимание на место для двух групповых символов, обозначенных знаками вопроса. Позже, при вызове метода `execute()`, эти заполнители будут заменены реальными значениями.

Полагая, что имеется массив `$items`, содержащий штрих-коды и идентификаторы продуктов, мы хотим осуществить одну вставку в базу данных для каждого товара. Строим цикл, выполняющийся для каждого элемента массива `$items`, извлекая штрих-код и идентификатор продукта, а затем исполняя подготовленный оператор SQL.

Как уже говорилось, метод `execute()` заменит заполнители в подготовленном операторе значениями, переданными ему во втором аргументе в форме массива. В обсуждаемом примере это будет аргумент `array($barcode, $item_id)`. Значения-заполнители заменяются в том порядке, в котором задаются новые значения, поэтому важно указывать их правильно.

Надеемся, что рассмотренное свойство PEAR DB-интерфейса окажется весьма полезным для ваших проектов.

Резюме

В главе представлены различные способы доступа к базе данных PostgreSQL из языка сценариев PHP.

Описаны разнообразные аспекты соединений с базой данных, построение и выполнение запросов, манипулирование результирующими множествами и обработка ошибок. Мы также познакомились с абстрактным интерфейсом базы данных PEAR.

Думаем, что основы заложены, и вы изучили достаточный набор базовых инструментов, с которым можно приступать к разработке собственных веб-ориентированных приложений.

16

Доступ к PostgreSQL из Perl

В предыдущих главах было показано, что взаимодействие с PostgreSQL, как правило, подразумевает массу работы с символьными строками. Язык, обладающий выдающимися возможностями обработки строк, – это Perl. В данной главе рассматривается совместное использование Perl и PostgreSQL.

В главе 13 говорилось о том, что возможности интерфейса `libpq` велики, но очевидно, что представление символьных строк, присущее языку C, достаточно примитивно. Последовательности символов завершаются значением `NULL`, и в небольших программах код, обрабатывающий строки, может оставить в тени взаимодействие с базой данных. В главе 13 говорилось также, что хотя и возможен бинарный доступ, выгода от этого минимальна. В Perl возможности строки гораздо шире, они поддерживают такие функциональности, как соединение, разбиение на части, поиск по шаблону и автоматическое преобразование из других типов данных и в другие типы.

Perl также исторически связан с обработкой данных на веб-серверах, а наличие интерфейсов к базам данных является дополнительным преимуществом. Правда, в настоящее время на арену выходят более современные механизмы, такие как язык PHP, описанный в предыдущей главе.

Данная глава не ставит перед собой цель обучить читателя языку Perl. Для тех, кто совсем с ним не знаком, приведем несколько полезных источников, с которых можно начать:

- *http://www.perl.org* и *http://www.cpan.org*
- «Основы Perl» («Beginning Perl»), Саймон Казенс (Simon Cozens), Wrox Press (ISBN 1-861003-14-5)

- «Изучаем Perl» («Learning Perl»), Рэндал Л. Шварц (Randal L. Schwartz) и Том Кристиансен (Tom Christiansen), O'Reilly (ISBN 1-56592-284-0)

Код, представленный в этой главе, не использует многие идиомы Perl, так что он должен быть понятен большинству программирующих на C. Необходимо помнить следующее:

- Скалярные переменные (числа или строки, Perl по требованию преобразовывает одни в другие) начинаются с символа `$`
- Списки (простые массивы) начинаются с символа `@`
- Хеши (ассоциативные массивы) начинаются с символа `%`

Новичков может смутить то обстоятельство, что символы `@` и `%` применяются только для ссылки на всю совокупность, а на отдельный элемент следует ссылаться посредством символа `$`.

Те, кто хотя бы немного знаком с Perl, должны знать, что в соответствии с одной из его аксиом каждую поставленную задачу можно решить несколькими способами. И в самом деле, энтузиасты Perl были бы разочарованы, если бы им пришлось ограничиться единственной возможностью. Однако мы не собираемся донимать вас многочисленными техниками доступа к базам данных PostgreSQL из Perl, вместо этого представим два лучших механизма.

По существу, есть три способа доступа к PostgreSQL из Perl:

- Низкого уровня – Perl-аналог C-интерфейса `libpq` (`pgsql_perl5`)
- Высокого уровня – независимый от базы данных
- Встраиваемый интерпретатор Perl (подобно встроенному SQL, описанному в главе 14)

Будет представлен пример низкоуровневого доступа, поскольку он очень близок к интерфейсу C, рассмотренному в главе 13. Поговорим и о высокоуровневом доступе, т. к. он очень гибок и мощен. А вот третий возможный вариант опустим. В применении PL/Perl не так много преимуществ, как, скажем, в `ecpg` для C, ведь обработка строк в Perl гораздо проще, чем в C. Кроме того, собирать PL/Perl сложно, т. к. ему необходима версия Perl, построенная как разделяемая библиотека (`libperl.so` вместо более привычной `libperl.a`), что выходит за рамки данной книги (см. инструкцию по сборке Perl в дистрибутиве с исходными текстами).

Модуль pgsql_perl5 или Pg

Существует несколько модулей Perl, предоставляющих прямой (или настолько прямой, насколько Perl это позволяет) доступ к функциональности `libpq`, одним из которых является модуль Эдмунда Мергла (Edmund Mergl) `pgsql_per15`, называемый также `Pg`. Именно этот мо-

дуль, вероятно, будет первым, на который вы наткнетесь, хотя бы потому, что он входит в дистрибутив PostgreSQL.

Установка pgsql_perl5

Веб-сайт PostgreSQL содержит несколько подготовленных для разных операционных систем пакетов, загрузив которые, вы обеспечиваете самый простой способ установки модуля. Например, если вы работаете в RedHat Linux, просто выберите:

```
postgresql-perl-version-release.architecture.rpm
```

установите его с помощью одной из графических утилит RPM или же введя такую команду:

```
$ rpm -i$ rpm -i postgresql-perl-7.1.2-1.i386.rpm
```

Однако может случиться и так, что вы будете вынуждены (или просто предпочтете) заняться установкой из исходных текстов. При наличии полного дерева исходных текстов PostgreSQL можно как присоединить интерфейсы Perl в процессе сборки, так и работать с ними по отдельности. Чтобы реализовать первый вариант, добавьте параметр `--with-perl` в конец команды `configure` (первый шаг в построении PostgreSQL, см. главу 3), а затем постройте полный дистрибутив PostgreSQL, как обычно.

Если же вы предпочитаете собрать этот модуль Perl отдельно, например, если имеется более новая версия (версия 1.8.0 включена в PostgreSQL 7.1.2, но в CPAN на момент написания книги можно было обнаружить версию 1.9.0), тогда распакуйте архив:

```
$ tar xfz pgsql_perl5-1.9.0.tar.gz
```

Прежде чем приступать к сборке, укажите в `POSTGRES_INCLUDE` каталог, содержащий заголовочные файлы `libpq` (такие как `/usr/local/pgsql/include` в стандартном дистрибутиве Red Hat), а в `POSTGRES_LIB` – каталог с библиотечными файлами (например, `/usr/local/pgsql/lib`). Обратите внимание, что описывается процедура сборки версии 1.9.0: версия 1.8.0 использует вместо этого одну переменную окружения `POSTGRES_HOME`, которая должна указывать на каталог, в котором установлена PostgreSQL (так, чтобы включаемые файлы PostgreSQL находились в `$POSTGRES_HOME/include`, а библиотеки – в `$POSTGRES_HOME/lib`). Затем выполните следующие команды:

```
$ perl Makefile.PL
$ make
$ make test
```

И последний шаг, который должен быть выполнен от имени суперпользователя:

```
# make install
```

CPAN

CPAN (Comprehensive Perl Archive Network) – это основное хранилище фактически всех существующих модулей Perl, расположенное по адресу *http://www.cpan.org*. Этот ресурс хорошо знаком практически всем Perl-программистам.

Последовательность команд, представленная выше, применяется для всех модулей CPAN: этап конфигурирования, порождающий настоящий make-файл, который затем может быть собран. Третий этап, на котором проверяется успешность сборки, не обязателен, но полезен. И на завершающем шаге построенные файлы копируются на свои правильные места в файловом пространстве операционной системы.

Последовательность сборка–установка настолько унифицирована, что существует удобное краткое обозначение. Если какой-то модуль CPAN уже был установлен (один раз **придется** сделать это вручную), то дальше можно загружать, собирать и устанавливать любой модуль CPAN, используя, например:

```
perl -MCPAN -e 'install DBI'
```

Те, кто работает с версией ActiveState Perl (*http://www.activestate.com*), используют подобную краткую запись в команде `ppm`. Если вам интересны подробности, выполните команду `ppm -h`.

Кстати, модули CPAN принимают и другие условные обозначения помимо последовательности сборки, например для документации, которая может быть просмотрена с помощью команды `perldoc` после установки модуля:

```
perldoc DBI
```

Применение pgsql_perl5

Интерфейс `pgsql_perl5` очень похож на интерфейс C, поэтому названия функций должны быть вам хорошо знакомы. На самом деле интерфейс Perl настолько близок к C, что нижеследующее практически повторяет программу `select2.c` из главы 13:

```
select_c.pl
#!/usr/bin/perl -w

use Pg;
use strict;

sub doSQL
{
    my ($conn, $command) = @_;
```

```
    print $command, "\n";

    my $result = $conn->exec($command);
    print "status is ", $result->resultStatus, "\n";
    print "#rows affected ", $result->cmdTuples, "\n";
    print "result message: ", $conn->errorMessage, "\n";

    if($result->resultStatus eq PGRES_TUPLES_OK) {
        print "number of rows returned = ", $result->ntuples, "\n";
        print "number of fields returned = ", $result->nfields, "\n";

        for(my $r = 0; $r < $result->ntuples; ++$r) {
            for(my $n = 0; $n < $result->nfields; ++$n) {
                print " ", $result->fname($n), " = ",
                      $result->getvalue($r, $n), "(",
                      $result->getlength($r, $n), "),";
            }
            print "\n";
        }
    }
}

my $conn = Pg::connectdb("");

if($conn->status eq PGRES_CONNECTION_OK) {
    print "connection made\n";

    doSQL($conn, "DROP TABLE number");
    doSQL($conn, "CREATE TABLE number ( value INTEGER, name  VARCHAR )");
    doSQL($conn, "INSERT INTO number values(42, 'The Answer')");
    doSQL($conn, "INSERT INTO number values(29, 'My Age')");
    doSQL($conn, "INSERT INTO number values(29, 'Anniversary')");
    doSQL($conn, "INSERT INTO number values(66, 'Clickety-Click')");
    doSQL($conn, "SELECT * FROM number WHERE value = 29");
    doSQL($conn, "UPDATE number SET name = 'Zaphod' WHERE value = 42");
    doSQL($conn, "DELETE FROM number WHERE value = 29");
} else {
    print "connection failed\n";
}
```

В действительности это **новый стиль**. Существует и старый интерфейс, который еще более родствен `libpq`. Например, вместо `$conn->exec($command)` старый интерфейс использует `PQexec($conn, $command)`. Однако новый стиль лучше вписывается в Perl. Старый же будет удален из следующих версий модуля.

Для запуска сценария применяем такую команду:

```
$ perl select_c.pl
```

Заметьте, что как и в C-версии, из которой это было перенесено, здесь используются переменные окружения PG* или же осуществляется соединение с базой данных по умолчанию.

Есть и другая возможность: если пометить сценарий как исполняемый (с помощью chmod 750 select_c.pl), то можно просто ввести:

```
$ ./select_c.pl
```

Вывод почти идентичен тому, который порождается соответствующим кодом C:

```
connection made
DROP TABLE number
status is 1
#rows affected is
error message:
CREATE TABLE number ( value INTEGER, name  VARCHAR )
status is 1
#rows affected is
error message:
INSERT INTO number values(42, 'The Answer')
status is 1
#rows affected is 1
error message:
INSERT INTO number values(29, 'My Age')
status is 1
#rows affected is 1
error message:
INSERT INTO number values(29, 'Anniversary')
status is 1
#rows affected is 1
error message:
INSERT INTO number values(66, 'Clickety-Click')
status is 1
#rows affected is 1
error message:
SELECT * FROM number WHERE value = 29
status is 2
#rows affected is
error message:
number of rows returned = 2
number of fields returned = 2
 value = 29(2), name = My Age(6),
 value = 29(2), name = Anniversary(11),
UPDATE number SET name = 'Zaphod' WHERE value = 42
status is 1
#rows affected is 1
error message:
DELETE FROM number WHERE value = 29
status is 1
#rows affected is 2
error message:
```

В этом коде обратим внимание на операции, присущие PostgreSQL. Первая представляющая интерес строка – это эквивалент директивы `#include <libpq-fe.h>` C, функции модуля становятся доступны нашей программе:

```
use Pg;
```

Оставим `doSQL` и обратимся к основной программе. Соединение с базой данных осуществляется посредством команды:

```
$conn = Pg::connectdb("");
```

Как и для `PQconnectdb()`, в качестве аргумента задается строка соединения, например `dbname=bpsimple user=rick`, и те же самые переменные окружения применяются для значений, не заданных явно (табл. 16.1):

Таблица 16.1. Переменные окружения

Данные	Переменные окружения	По умолчанию
Сервер базы данных	PGHOST	"localhost"
Номер порта	PGPORT	5432
Параметры	PGOPTIONS	Пустая строка
База данных	PGDATABASE	template1
Пользователь	PGUSER	Идентификатор пользователя
Пароль	PGPASSWORD	Пустая строка

Переменная `$conn` – это дескриптор соединения, соответствующий указателю на `Pgconn` в C. Значениями статуса выполнения в Perl являются не числа, как в C, а простые строки, поэтому их следует сравнивать при помощи `eq`, а не `==`. Модуль `pgsql_perl5` определяет ряд строковых констант, таких как `PGRES_EMPTY_QUERY`, `PGRES_COMMAND_OK` и `PGRES_TUPLES_OK`.

Если соединение успешно установлено, то `doSQL` запускается столько раз, сколько необходимо для выполнения операций с базой данных.

Сборщик мусора Perl занимается уборкой в конце программы, аналога функции `PQfinish()` не существует, однако можно указать, что доступ к базе данных больше не требуется, установив `$conn` в какое-нибудь другое значение, например `undef`. При этом остается риск, что соединение с базой данных будет открыто дольше, чем если бы было произведено явное отсоединение.

Внутри функции `doSQL` запрос или команда выполняются так:

```
$result = $conn->exec($command);
```

`$result` – это дескриптор результата, соответствующий `PGresult*` в C. Дескриптор результата предоставляет доступ к таким привычным полям, как `resultStatus`, `cmdStatus`, `ntuples` и `nfields`. Обратите внимание, что в `pgsql_perl5` нет доступа к сообщениям об ошибках, поэтому при-

ходится ограничиться информацией о соединении (`$conn->errorMessage`). В данной программе это единственное существенное отличие между прямыми интерфейсами C и Perl к PostgreSQL.

Информация об отдельных полях может быть получена так:

```
$fname   = $result->fname($fieldNum);
$fnumber = $result->fnumber($fieldName);
$ftype   = $result->ftype($fieldNum);
$fsize   = $result->fsize($fieldNum);
```

Полная идентичность аналогам из `libpq`: `$fieldNum` – это номер столбца (нумерация начинается с нуля), а `$fieldName` – это название столбца. Результаты запросов можно получить таким образом:

```
$value  = $result->getvalue($tupleNum, $fieldNum);
$length = $result->getlength($tupleNum, $fieldNum);
$isNull = $result->getisnull($tupleNum, $fieldNum);
```

И снова все подобно операциям `libpq`, имеющим те же названия, при этом `$tupleNum` – это номер строки в результате запроса (нумерация начинается с нуля).

Отсутствует явная операции очистки – нет эквивалента `PQclear()`. Это не ошибка: сборщик мусора Perl занимается очисткой позже, как и отсоединением от базы данных.

Как уже говорилось, налицо очень близкое соответствие примеру на C из главы 13. Это больше похоже на подражание C в Perl, чем на подлинный код Perl. Например, вложенный цикл `for` в середине программы представляет собой полную копию примера на C, но он не очень свойствен Perl. Более подходящая идиома приведена ниже, в ней используется функция `fetchrow` модуля `pgsql_perl5`, которая возвращает следующую строку результатов запроса в виде массива (и `NULL` в конце):

```
while(my @row = $result->fetchrow) {
    print " ", join(" ", @row), "\n";
}
```

Функция `join` просто соединяет все элементы массива вне зависимости от их размера, используя заданный разделитель. Если же известно количество столбцов, можно применить другой прием Perl, массив переменных в левой части присваивания:

```
($number, $value) = $result->fetchrow
```

Здесь `$number` и `$value` – это пара переменных, которым будут присвоены значения первого и второго столбцов.

Приведем теперь ту же самую программу, написанную в манере, характерной для Perl. Именно этот сценарий будет базовым компонентом для других фрагментов кода Perl, представленных в данной главе,

так мы сможем более понятно показать изменения, которые необходимо вносить для реализации различных механизмов доступа к базе данных:

```
select.pl
#!/usr/bin/perl -w

use Pg;
use strict;

# Функция для не-запросов
sub doSQL
{
    my ($conn, $command) = @_;

    print $command, "\n";

    my $result = $conn->exec($command);
    print "status is ", $result->resultStatus, "\n";
    print "#rows affected ", $result->cmdTuples, "\n";
    print "result message: ", $conn->errorMessage, "\n";
}

# Функция специально для запросов
sub doSQLquery
{
    my ($conn, $command) = @_;

    print $command, "\n";

    my $result = $conn->exec($command);
    print "status is ", $result->resultStatus, "\n";

    return if($result->resultStatus ne PGRES_TUPLES_OK);

    print "number of rows returned = ", $result->ntuples, "\n";
    print "number of fields returned = ", $result->nfields, "\n";

    print "fields: ";
    for(my $n = 0; $n < $result->nfields; ++$n) {
       print " ", $result->fname($n);
    }
    print "\n";

    while(my @row = $result->fetchrow) {
         print " ", join(" ", @row), "\n";
    }
}
```

```
my $conn = Pg::connectdb("") or die "connection failed";

doSQL($conn, "DROP TABLE number");
doSQL($conn, "CREATE TABLE number ( value INTEGER, name  VARCHAR )");
doSQL($conn, "INSERT INTO number values(42, 'The Answer')");
doSQL($conn, "INSERT INTO number values(29, 'My Age')");
doSQL($conn, "INSERT INTO number values(29, 'Anniversary')");
doSQL($conn, "INSERT INTO number values(66, 'Clickety-Click')");
doSQLquery($conn, "SELECT * FROM number WHERE value = 29");
doSQL($conn, "UPDATE number SET name = 'Zaphod' WHERE value = 42");
doSQL($conn, "DELETE FROM number WHERE value = 29");
```

Для прекращения выполнения сценария в случае ошибки соединения применяется функция Perl `die`, сценарий становится более лаконичным, это вполне подходит для задач быстрого программирования, для которых Perl обычно и применяется. Еще одно важное отличие от предыдущего сценария заключается в том, что функция обработки SQL разбита на две, одна из которых используется для запросов, а другая – для команд, не содержащих запрос (зачем так сделано, станет понятно чуть позже).

Прежде чем закончить с `pgsql_perl5`, упомянем о том, что в Perl существуют вариации на тему `PQprint()` и асинхронной обработки, описанной в главе 13 (а также многих других возможностей `libpq`), но они настолько похожи на своих двойников из C, что вам будет достаточно материалов главы 13 и документации по `pgsql_perl5`.

Perl DBI

Тем, кто занимался программированием баз данных в Windows, должен быть знаком ODBC (Open Database Connectivity – открытый интерфейс доступа к базам данных) или более современные API, такие как ADO или OLE DB. Тем же, кто использовал Java с базами данных, наверняка встречался JDBC (см. следующую главу). Эти программные интерфейсы реализуют попытку абстрагироваться от деталей используемой в конкретный момент базы данных и предоставить интерфейс верхнего уровня, независимый от базы данных. Преимущество заключается в том, что, изучив всего один интерфейс, можно использовать свои приложения с различными базами данных.

Интерфейс базы данных (Database Interface, DBI) – это Perl-вариант такой схемы. Как и другие API, независимые от БД, DBI состоит из клиентского модуля API и одного или нескольких драйверов или модулей драйвера базы данных (DBD-модулей). Можно открывать одновременно несколько баз данных и получать к ним доступ, используя, по существу, один и тот же код в сценариях Perl (рис. 16.1):

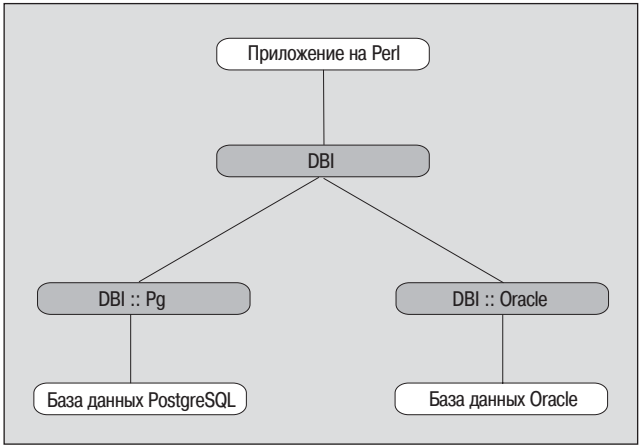


Рис. 16.1. Работа Perl-приложения с несколькими базами данных

Интересно, что один из доступных DBD – это драйвер ODBC – интерфейса, независимого от базы данных. Что касается ODBC, есть еще модуль `Win32::ODBC`, который также может применяться для соединения с PostgreSQL, но он не обсуждается в данной книге.

Подробная информация о DBI представлена на его домашней странице *http://www.symbolstone.org/technology/perl/DBI/* и в книге «Программирование на Perl DBI» («Programming the Perl DBI») Аллигатора Декарта (Alligator Descartes) и Тима Банса (Tim Bunce), O'Reilly (ISBN 1-56592-699-4).[1]

Установка DBI и PostgreSQL DBD

Как всегда, первым делом следует установить модули – сам DBI и необходимые DBD. Как обычно загрузите пакет из CPAN, соберите и установите (или используйте краткую запись, приведенную ранее). Вероятно, вы увидите множество предупреждений компилятора о неиспользуемых переменных. Это некрасиво, но безвредно.

Сам пакет DBI содержит несколько модулей DBD (ADO, Multiplex, Proxy и ExampleP, шаблон с очень ограниченными возможностями, предназначенный в основном для разработчиков DBD), но не драйвер для PostgreSQL, так что нужно разыскать его самостоятельно. На помощь опять приходит Эдмунд Мергл (Edmund Mergl) с модулем `DBD::Pg`. Загрузите его из CPAN, укажите в переменной окружения `POSTGRES_INCLUDE` каталог, содержащий заголовочные файлы `libpq`, а в `POSTGRES_LIB` – каталог с библиотечными файлами, затем выполните обычные шаги сборки и установки.

[1] А. Декарт, Т. Банс, «Программирование на Perl DBI», издательство «Символ-Плюс», 2000 г.

На секунду отвлечемся от PostgreSQL и посмотрим на еще один полезный драйвер, DBD::CSV. Он эмулирует доступ к базам данных, используя текстовые файлы значений, разделенных запятыми (Comma Separated Values, CSV). Он удобен для быстрого создания прототипов сценариев, т. к. не требует установки реального сервера базы данных.

Использование DBI

Можно ожидать, что независящий от базы данных уровень имеет не такой программный интерфейс приложения, как использовавшийся до этого специальный API PostgreSQL. В действительности, поскольку цель DBI заключается в том, чтобы обеспечить абсолютную независимость от базы данных, вы обнаружите, что просто невозможно достичь такой краткости и эффективности, как в случае применения специального интерфейса к PostgreSQL, такого как pgsql_perl5.

DBI имеет некоторую дополнительную функциональность, о которой поговорим после того, как посмотрим на предыдущий сценарий, переделанный для использования DBI; измененные строки выделены:

```
select_dbi.pl
#!/usr/bin/perl -w

use DBI;
use strict;

# Функция для незапросов
sub doSQL
{
    my ($conn, $command) = @_;

    print $command, "\n";

    my $sth = $conn->prepare($command);
    my $nrows = $sth->execute;
    print "status is ", $DBI::err, "\n" if $DBI::err;
    print "#rows affected is ", $nrows, "\n";
    print "error message: ", $DBI::errstr, "\n" if $DBI::err;
}

# Функция специально для запросов
sub doSQLquery
{
    my ($conn, $command) = @_;

    print $command, "\n";

    my $sth = $conn->prepare($command);
    my $nrows = $sth->execute;
    print "status is ", $DBI::err, "\n" if $DBI::err;

    print "number of rows returned (unreliable) = ", $sth->rows, "\n";
    print "number of fields returned = ", $sth->{NUM_OF_FIELDS}, "\n";
```

```
        print "fields: ", join(" ", @{$sth->{NAME}}), "\n";

        while(my @row = $sth->fetchrow_array) {
            print " ", join(" ", @row), "\n";
        }
    }

    my $conn = DBI->connect("DBI:Pg:") or die $DBI::errstr;

    doSQL($conn, "DROP TABLE number");
    doSQL($conn, "CREATE TABLE number ( value INTEGER, name  VARCHAR )");
    doSQL($conn, "INSERT INTO number values(42, 'The Answer')");
    doSQL($conn, "INSERT INTO number values(29, 'My Age')");
    doSQL($conn, "INSERT INTO number values(29, 'Anniversary')");
    doSQL($conn, "INSERT INTO number values(66, 'Clickety-Click')");
    doSQLquery($conn, "SELECT * FROM number WHERE value = 29");
    doSQL($conn, "UPDATE number SET name = 'Zaphod' WHERE value = 42");
    doSQL($conn, "DELETE FROM number WHERE value = 29");

    $conn->disconnect;
```

Первое, на что нужно обратить внимание, – упоминается **только** модуль DBI. Не надо явно импортировать специальные драйверы для той конкретной базы данных, с которой вы работаете:

```
use DBI;
```

После того как соединение с базой данных установлено (см. далее), DBI определит местоположение соответствующего DBD и загрузит его (если его еще нет) или же выдаст ошибку.

В основной программе база данных открывается с помощью:

```
$conn = DBI->connect("DBI:Pg:");
```

Первый аргумент соединения – это источник данных, который должен начинаться с `DBI`, а затем должно стоять название драйвера. Оставшаяся часть строки специфична для DBD, вся же строка источника данных выглядит так:

```
DBI:⟨имя драйвера⟩:⟨параметры драйвера⟩
```

Для DBI регистр не важен, но название драйвера чувствительно к регистру, а остаток строки зависит от правил данного драйвера. Для PostgreSQL параметры драйвера аналогичны строке, передаваемой `PQconnectdb()`, например `dbname=bpsimple`; они могут быть опущены, тогда будут использованы переменные окружения (объяснялось при описании `pgsql_perl5`).

DBI вводит еще один уровень переменных окружения; если часть названия драйвера пуста (то есть между первым и вторым двоеточием ничего нет, как в `DBI::dbname=bpsimple`), то DBI берет имя DBD из переменной окружения `DBI_DRIVER`. Наконец, для удобства может быть опу-

щена или оставлена пустой строка источника данных, тогда DBI подставляет вместо нее переменную окружения DBI_DSN. В сценарии представлена краткая форма функции connect, автоматически отыскивающая идентификатор пользователя и пароль в переменных окружения DBI_USER и DBI_PASS, если они существуют. Полная же форма такова:

```
$dbh = DBI->connect($dsn, $uname, $pwd, \%attrs);
```

Вторым и третьим аргументами являются идентификатор пользователя и пароль, с которыми осуществляется доступ в базу данных, а последний, необязательный аргумент – это хеш, содержащий атрибуты, подлежащие передаче DBI.

Есть такой важный атрибут, как AutoCommit, – установка его в 1 приводит к тому, что DBI рассматривает каждую операцию как отдельную транзакцию, если же установить его в 0, придется явно обрабатывать транзакции.

Приведем пример полного задания соединения:

```
$dbh = DBI->connect("DBI:Pg:dbname=bpsimple",
                    "rick", "",
                    { AutoCommit => 0 });
```

В данном примере автоматическое завершение выключено, поэтому для завершения транзакций необходимо добавлять вызовы $dbh->commit (или $dbh->rollback).

Как уже говорилось, имя пользователя может быть указано или же получено из $DBI_USER или даже $PGUSER. То есть существует несколько способов определения параметров соединения с базой данных. Надо ли последовательно применять один механизм, чтобы не перепутать, какой из них имеет приоритет над остальными?

Возвращаемое connect значение – это дескриптор соединения с базой данных или undef в случае ошибки. Если произошла ошибка, то соответственно устанавливаются $DBI::err и $DBI::errstr. В приведенном выше коде в случае ошибки обработка просто завершается, и выводится сообщение об ошибке посредством функции Perl die.

Когда обработка завершена, соединение с базой данных закрывается:

```
$conn->disconnect;
```

Хотя соединение будет закрыто автоматически при выходе из приложения (при этом будет выведено предупреждение), рекомендуется явно закрывать соединение, чтобы избежать проблем с незавершенными транзакциями. Если одновременно открыто несколько соединений, можно закрыть их все сразу при помощи DBI->disconnect_all.

Вернемся к функциям doSQL. DBI разбивает операцию на два шага (три для запроса): подготовка и выполнение (и выборка результатов –

для запроса). Дело в том, что может потребоваться выполнить одну и ту же операцию несколько раз, и тогда сервер базы данных позволит оптимизировать доступ, подготовив операцию один раз и выполняя ее многократно:

```
my $sth = $conn->prepare($command);
my $nrows = $sth->execute;
```

В результате подготовки появляется дескриптор оператора, применяемого для обработки, а в результате выполнения мы получаем количество измененных строк или `undef`, если произошла ошибка. В некоторых случаях, когда требуется выполнить оператор всего один раз, можно объединить два шага в одном вызове функции, тогда строка, приведенная ниже, заменит две:

```
my $nrows = $conn->do($command);
```

Заметьте, что здесь нет дескриптора оператора, а это означает, что невозможно извлечь результат запроса, поэтому `do` можно использовать только для команд, не содержащих запросы.

Следующее, на что следует обратить внимание: в то время как `pgsql_perl5` позволяет применять одну и ту же обработку как для запросов, так и для операторов SQL, запросами не являющихся, DBI этого не разрешает, потому что некоторые базы данных используют разные механизмы для этих двух операций. Именно по этой причине общая функция `doSQL` разделена на две: для запросов и не-запросов. Кроме того, поскольку некоторые базы данных не могут сообщить о количестве строк, возвращенных запросом, они просто возвращают строки до тех пор, пока те не закончатся. Строго говоря, значение, возвращаемое функцией `execute`, ненадежно, но для PostgreSQL оно всегда корректно.

Обработка результатов запроса аналогична предыдущему примеру. После того как оператор выполнен, для дескриптора оператора используется функция `fetchrow_array`. Существуют и другие варианты этой функции:

- `fetchrow_arrayref`

  Возвращает ссылку на внутренний массив (которая заново используется при следующей выборке), благодаря чему избегает копирования, выполняемого функцией `fetchrow_array` несмотря на то, что приложение должно использовать данные незамедлительно.

- `fetchrow_hashref`

  Возвращает ссылку на хэш с названиями столбцов в качестве ключей и данными в качестве соответствующих значений.

- `fetchall_arrayref`

  Возвращает все результаты запроса в ссылке на единый массив. Кстати, тем самым обеспечивается переносимый и надежный спо-

соб определения количества строк, но если их очень много, то это приводит к чрезмерному расходованию памяти.

Функция `do` для операторов, не являющихся запросами, уже упоминалась ранее. Существует похожая краткая запись или, если быть точным, то набор записей для запросов, о которых известно, что их не надо будет использовать повторно: `selectrow_array` и `selectall_arrayref` осуществляют подготовку и выполнение операций, за которыми (внутри одного вызова) следует соответствующая функция выборки. Подробности об этих и других вариантах запросов, доступных в DBI, представлены в документации.

После завершения работы с дескриптором оператора следует освободить его, установив переменную в `undef`, или просто выйти из его области видимости. Можно также вызвать:

```
$sth->finish;
```

Такой же вызов необходимо осуществить, если вы хотите повторно использовать запрос, для того чтобы сбросить буфер результатов, иначе вы рискуете опять получить строки из предыдущего запроса. О заполнителях поговорим чуть позже в этой главе.

Мы уже знаем, что DBI делит операции на два этапа: подготовку и исполнение, чтобы можно было выполнять запрос несколько раз. Зачем это может потребоваться? Не будем ли мы все время получать одни и те же результаты, если никакой другой процесс не будет менять базу данных? Это справедливо за одним лишь исключением для небольшой особенности функционирования DBI – возможности связывать параметры.

Вместо того чтобы вводить полностью заданную строку как SQL-оператор, можно использовать в ней в качестве заполнителей знаки вопроса, а реальные значения будут поставлены им в соответствие позже (аналогично синтаксису переменных основного языка в `ecpg`, описанному в главе 14). Подготавливаем оператор один раз и выполняем его с различными значениями, указывая по аргументу для каждого заполнителя. Приведенный ниже пример вернет строки со значениями 14 и 15:

```
my $sth = $conn->prepare("SELECT * FROM number WHERE value = ?");
$sth->execute(14);
# Обработка строк...
$sth->execute(15);
# Обработка строк...
```

Каждый знак вопроса (заполнитель) в операторе SQL заменяется соответствующим аргументом в `execute`. Можно записать `$sth->execute(14)` следующим образом:

```
$sth->bind_param(1, 14);  # Замечание: нумерация параметров начинается с 1
$sth->execute;
```

Особой пользы не видно, зато больше приходится набирать на клавиатуре. Однако если в операторе много заполнителей, а от выполнения к выполнению меняются лишь некоторые из них, это может быть удобно. DBI также поддерживает связывание по ссылке, которое, хотя и предназначено для передачи значений в хранимые процедуры и обратно, может несколько уменьшить объем ввода с клавиатуры:

```
my $num;
my $sth = $conn->prepare("SELECT * FROM number WHERE value = ?");
$sth->bind_param_inout(1,\$num, 10);
$num = 14;
$sth->execute;
# Обработка строк...
$num = 15;
$sth->execute;
# Обработка строк...
```

Здесь используется ссылка, поэтому любые изменения значения переменной сразу же и без особых усилий становятся доступны для DBI. Необходим еще третий аргумент, указывающий максимальный размер возвращаемого значения; в данном конкретном случае он неуместен, но, тем не менее, должен быть указан.

Связывание работает и для результатов запросов. Здесь результаты обрабатываются без использования `doSQLquery`:

```
sub doSQLquery
{
    my ($conn, $command) = @_;

    print $command, "\n";

    my $sth = $conn->prepare($command);
    my $nrows = $sth->execute;
    print "status is ", $DBI::err, "\n" if $DBI::err;

    print "number of rows returned (unreliable) = ", $sth->rows, "\n";

   my( $name, $value );
    $sth->bind_col(1, \$value); # 1st column mapped on to $value
    $sth->bind_col(2, \$name);  # 2nd column mapped on to $name

    while($sth->fetch) {
        print " name = ", $name, ", value = ", $value, "\n";
    }
}
```

Связывание может несколько увеличить производительность (зависит от используемой базы данных), но оно безусловно улучшает понятность кода.

Что еще можно сделать с помощью DBI?

Пока что рассматривалось в основном специальное применение DBI для PostgreSQL, теперь же посвятим немного времени некоторым его дополнительным возможностям, не зависящим от базы данных.

Можно без труда получить список доступных драйверов и, в некоторых случаях, баз данных, что показано в нижеследующем сценарии:

```
dbi_sources.pl
#!/usr/bin/perl -w

use DBI;
use strict;

foreach my $driver (DBI->available_drivers())
{
    print "Driver ", $driver;

    eval { print "\n", join("\n ", DBI->data_sources($driver)), "\n\n" };
    print " - error ", $@, "\n\n" if ($@);
}
```

Функция `DBI->available_drivers` возвращает список всех DBD-драйверов, правда, не проверяя, можно ли их применить или по крайней мере загрузить. Для некоторых DBD можно определить, какие базы данных доступны, используя `DBI->data_sources` – эта функция пытается загрузить драйвер, что по какой-то причине может не удаться, поэтому она помещена в блок `eval`. Возвращаемое `eval` значение представляет собой выражение, находящееся внутри блока. Статус ошибки находится в переменной `$@`, что вызывает необходимость ее проверки на неравенство `NULL`. На одном из наших компьютеров результат работы сценария выглядит так:

```
Driver ADO

Driver ExampleP
 dbi:ExampleP:dir=.

Driver Multiplex

Driver Pg
 dbi:Pg:dbname=bpsimple
 dbi:Pg:dbname=pgperltest
 dbi:Pg:dbname=template1

Driver Proxy
```

Если выполнить то же самое, не используя `eval`, то будет выведено сообщение об ошибке и сценарий преждевременно завершится:

```
install_driver(Proxy) failed: Can't locate RPC/PlClient.pm in @INC (@INC
contains: /usr/lib/perl5/5.6.0/i386-linux /usr/lib/perl5/5.6.0 /usr/lib/
perl5/site_perl/5.6.0/i386-linux /usr/lib/perl5/site_perl/5.6.0
```

```
/usr/lib/perl5/site_perl .) at /usr/lib/perl5/site_perl/5.6.0/i386-linux/
DBD/Proxy.pm line 28.
BEGIN failed--compilation aborted at /usr/lib/perl5/site_perl/5.6.0/
i386-linux/DBD/Proxy.pm line 28.
Compilation failed in require at (eval 5) line 3.
Perhaps a module that DBD::Proxy requires hasn't been fully installed
 at dbi_sources.pl line 10
```

Недвусмысленно сказано, что загрузка DBD для proxy не состоялась.

Ранее упоминался атрибут `AutoCommit`, приведем еще несколько интересных атрибутов:

- `PrintError`

  Как и заполнение `$DBI::errstr`, он приводит к выводу сообщений об ошибках на `stderr`

- `RaiseError`

  Вместо того чтобы просто возвратить статус, ошибки вызовут завершение программы (`die`), если не заключить вызов в блок `eval`

- `Name`

  Имя базы данных, обычно совпадающее со строкой, переданной для соединения

Первые два атрибута могут быть полезны для быстрого предварительного тестирования, но, как правило, они не слишком дружественны по отношению к конечным пользователям. Существуют и другие, менее распространенные атрибуты, описываемые в документации по DBI и PostgreSQL DBD.

Доступ к атрибутам можно получить после соединения, в некоторых случаях есть возможность изменить их, хотя и рекомендуется этого не делать. Дело в том, что не все DBD поддерживают динамические изменения, поэтому так можно ограничить переносимость сценария. Приведенный ниже пример считывает и устанавливает режим работы с транзакциями:

```
$oldState = $conn->{ AutoCommit };
$conn->{ AutoCommit } = 0;
```

Атрибуты сопоставлены не только дескрипторам базы данных, но и дескрипторам операторов. Ранее уже приводился такой пример:

```
print "number of fields returned = ", $sth->{NUM_OF_FIELDS}, "\n";
```

Перечень атрибутов операторов приведен в табл. 16.2:

Таблица 16.2. Атрибуты операторов

Атрибут	Описание
`NUM_OF_FIELDS`	Количество полей, возвращенных запросом
`NUM_OF_PARAMS`	Количество заполнителей

Таблица 16.2 (продолжение)

Атрибут	Описание
NAME	Ссылка на массив, содержащий названия столбцов
NAME_lc	Как и NAME, но всегда возвращает результат в нижнем регистре
NAME_uc	Как и NAME, но всегда возвращает результат в верхнем регистре
NULLABLE	Ссылка на массив, содержащий флаги, указывающие, может ли каждый столбец принимать значения NULL
Statement	Строка, используемая для создания оператора
TYPE	Ссылка на массив, указывающий тип каждого столбца

Знайте, что все эти атрибуты доступны только для чтения, и поскольку в зависимости от базы данных некоторые из них могут стать доступны лишь после выполнения оператора, то лучше использовать их только в этом случае.

Это лишь поверхностный обзор DBI, в оперативной справке вы найдете гораздо больше информации. Особого упоминания заслуживает оболочка DBI – dbish. Она похожа на psql, но, как и сам DBI, не зависит от базы данных.

Завершим раздел кратким сравнением двух механизмов доступа к базам данных PostgreSQL из Perl: DBI и pgsql_perl5. DBI обеспечивает независимость от базы данных и предоставляет достаточно изощренные методы обработки и связывания параметров. С другой стороны, pgsql_perl5 проще, и он предоставляет более прямой, а значит, и эффективный путь к базе данных. Кроме того, если вы пользовались libpq, то pgsql_perl5 будет вам более привычен.

Наконец, оба механизма включают в себя обработку больших бинарных объектов – «блобов» (см. приложение F). В pgsql_perl5 предоставляется комплексный подход к обработке, но независимость DBI от базы данных не позволяет ему обеспечить полную поддержку. Если предполагается, что приложение будет работать с несколькими базами данных, то единственное возможное решение – это DBI; а для быстрых сценариев одноразового употребления лучше простой pgsql_perl5.

Использование DBIx::Easy

Те, кто просматривал разделы CPAN, относящиеся к базам данных, должны были заметить множество модулей DBIx. Это разнообразные модули, усовершенствующие различные аспекты программирования. Особый интерес представляет один из них, DBIx::Easy (домашняя страница *http://www.linuxia.de/DBIx/Easy/*, хотя все, что нужно, можно найти и с помощью CPAN), простейший интерфейс к DBI. DBIx::Easy поддерживает ограниченное подмножество DBD. К счастью, драйвер PostgreSQL входит в их число.

В этом модуле некоторые операции теряют характерные для них черты. Решайте сами, хорошо это или не очень. Например, можно вернуть результаты запроса в виде хэша (строго говоря, ссылки на хэш). Есть и еще одна полезная возможность: не надо проверять значение, возвращаемое каждой операцией с базой данных, вместо этого можно установить обработчик ошибок.

Вот и опять наш вездесущий сценарий Perl, на этот раз использующий `DBIx::Easy`, в котором продемонстрированы обе возможности:

```
select_easy.pl
#!/usr/bin/perl -w

use DBIx::Easy;
use strict;
sub myErrorHandler
{
    my( $statement, $err, $msg ) = @_;
    die"Oops, \"$statement\" failed ($err) - $msg";
}

# Замечание: следует явно указать тип и имя базы данных
my $conn = new DBIx::Easy("Pg", "bpsimple");

$conn->install_handler(\&myErrorHandler);

$conn->process("DROP TABLE number");
$conn->process("CREATE TABLE number ( value INTEGER, name  VARCHAR )");
$conn->insert("number", name => "The Answer",     value => 42);
$conn->insert("number", name => "My Age",         value => 29);
$conn->insert("number", name => "Anniversary",    value => 29);
$conn->insert("number", name => "Clickety-Click", value => 66);

my $numbers = $conn->makemap("number", "name", "value", "value = 29");
foreach my $name (keys(%$numbers)) {
    print $name, " has value ", $$numbers{$name}, "\n";
}

$conn->update("number", "value = 42", name => "Zaphod");
$conn->process("DELETE FROM number WHERE value = 29");
$conn->disconnect;
```

Этот сценарий несколько отличается от предыдущих. Дело в том, что `DBIx::Easy` разделяет операции `process`, `insert`, `update` и `makemap` (запрос), поэтому было бы неуместно объединять их все в предыдущую пару функций `doSQL`.

Прежде чем приниматься за операции, посмотрим на обработчик ошибок: вместо того чтобы проверять каждую функцию на успешное выполнение или ошибку (к чему мы, вероятно, не очень аккуратно относились в предыдущих фрагментах кода), можно положиться на обра-

ботчик ошибок, вызываемый для каждой ошибки. В данном случае он обеспечивает только выход из сценария, но в своих программах вы можете задать для него более сложные действия.

Назначение `process`, `insert` и `update` достаточно очевидно, но `makemap` заслуживает пояснений. Эта функция принимает имя таблицы, названия двух столбцов и необязательную инструкцию `where`:

```
$conn->makemap($table, $keycol, $valuecol, $where)
```

На самом деле выполняется запрос:

```
SELECT $keycol, $valuecol FROM $table WHERE $where
```

Результаты вставляются в хэш так, как если бы это делал нижеприведенный код:

```
while(my ($key, $value) = $sth->fetchrow_array) {
    $map{$key} = $value;
}
```

Хэш может отобразить каждый ключ только в одно значение, поэтому многократные отображения исходной таблицы будут потеряны. Например, если таблица содержала две строки: (A, B) и (A, C), то только одна из них попадет в результирующий хэш, т. к. на первую, возвращенную базой данных, затем будет наложена вторая.

Есть и еще одно ограничение: одновременно могут обрабатываться лишь два столбца, в отличие от обычного запроса, который может возвращать любое заданное количество столбцов. Обратите внимание, что это не аналог хэша, возвращаемого функцией `fetchrow_hashref` в DBI. Данная функция возвращает одну строку, где ключом является название столбца, а значением – данные.

DBI и XML

Разговаривая о модулях `DBIx`, нельзя не отметить модуль Мэтта Сарджента (Matt Sergeant) `DBIx::XML_RDB`, упрощающий создание корректного (well-formed) XML из результатов запросов DBI.

Представим версию сценария Perl, на этот раз преобразующую результат запроса в XML (отличия от `select_dbi.pl` выделены):

```
select_xml.pl
#!/usr/bin/perl -w
use DBI;
use DBIx::XML_RDB;
use strict;

# Функция для не-запросов
sub doSQL
{
    my ($conn, $command) = @_;
```

```
    print $command, "\n";

    my $sth = $conn->prepare($command);
    my $nrows = $sth->execute;
    print "status is ", $DBI::err, "\n" if $DBI::err;
    print "#rows affected is ", $nrows, "\n";
    print "error message: ", $DBI::errstr, "\n" if $DBI::err;
}

# Функция специально для запросов
sub doSQLquery
{
    my ($conn, $command) = @_;

    print $command, "\n";

    $conn->DoSql($command);

    print $conn->GetData;
}

my $connXml = DBIx::XML_RDB->new("", "Pg") or die $DBI::errstr;
my $conn = $connXml->{dbh}
doSQL($conn, "DROP TABLE number");
doSQL($conn, "CREATE TABLE number ( value INTEGER, name  VARCHAR )");
doSQL($conn, "INSERT INTO number values(42, 'The Answer')");
doSQL($conn, "INSERT INTO number values(29, 'My Age')");
doSQL($conn, "INSERT INTO number values(29, 'Anniversary')");
doSQL($conn, "INSERT INTO number values(66, 'Clickety-Click')");
doSQLquery($connXml, "SELECT * FROM number WHERE value = 29");
doSQL($conn, "UPDATE number SET name = 'Zaphod' WHERE value = 42");
doSQL($conn, "DELETE FROM number WHERE value = 29");
```

База данных открывается следующим образом:

```
my $connXml = DBIx::XML_RDB->new("", "Pg") or die $DBI::errstr;
my $conn = $connXml->{dbh};
```

Функция `DBIx::XML_RDB->new` возвращает дескриптор соединения XML_RDB, отличающийся от дескриптора DBI. Изучение исходных текстов модуля показывает, однако, что он содержит дескриптор, который может быть использован для всех операций, не являющихся запросами (хотя, строго говоря, не стоит надеяться на это, и лучше не смешивать операции XML_RDB с другими операциями базы данных).

Операция, являющаяся запросом, `DoSql`, добавляет XML во внутреннюю строку, которую можно извлечь посредством `GetData`, что и показано в функции `doSQLquery`. Результаты нашего запроса таковы:

```
<?xml version="1.0"?>
<DBI driver="dbname=book">
    <RESULTSET statement="SELECT * FROM number WHERE value = 29">
        <ROW>
```

```
            <value>29</value>
            <name>My Age</name>
        </ROW>
        <ROW>
            <value>29</value>
            <name>Anniversary</name>
        </ROW>
    </RESULTSET>
</DBI>
```

Сам модуль `DBIx::XML_RDB` ограничивается порождением XML из запросов к базе данных, хотя пакет включает в себя пару сценариев для конвертирования таблицы в XML и обратно: `sql2xml.pl` использует способность запроса XML создавать дамп целой таблицы, а `xml2sql.pl` заново считывает ее оттуда.

Параметры первого из описанных сценариев представлены в табл. 16.3:

Таблица 16.3. Параметры sql2xml.pl

Параметр	Описание
`-sn servername`	Имя источника данных
`-driver dbi_driver`	Драйвер, используемый DBI (по умолчанию – ODBC)
`-uid username`	Имя пользователя
`-pwd password`	Пароль (необязательно)
`-table tablename`	Извлекаемая таблица
`-output outputfile`	Файл, в который будет помещен вывод XML

Если выполнить команду без параметров, то она выведет инструкцию по применению, идентифицировав себя как `sql2xls.pl`, и сошлется на файлы Excel, так что похоже, что документация не очень-то соответствует коду.

Можно выполнить сценарий для таблицы, созданной более ранними Perl-программами, с помощью команды (все в одной строке):

```
$ /usr/lib/perl5/site_perl/5.6.0/DBIx/sql2xml.pl
        -sn dbname=bpsimple -driver Pg
        -table number -otput xml.txt -uid rick
```

Более чем вероятно, что ваш путь не включает каталоги, содержащие эти сценарии, поэтому выше приведено указание полного пути (может понадобиться изменить его на конкретный каталог установки Perl). Выходной файл `xml.txt` содержит следующее:

```
<?xml version="1.0"?>
<DBI driver="dbname=bpsimple">
    <RESULTSET statement="SELECT * FROM number ORDER BY 1">
        <ROW>
            <value>42</value>
```

```
            <name>Zaphod</name>
        </ROW>
        <ROW>
            <value>66</value>
            <name>Clickety-Click</name>
        </ROW>
    </RESULTSET>
</DBI>
```

Как видите, формат практически идентичен примеру запроса, приведенному выше, а это показывает, что сценарий представляет собой достаточно тонкую «обертку» для модуля `DBIx::XML_RDB`.

Обратный сценарий, `xml2sql.pl`, имеет аналогичные параметры (табл. 16.4):

Таблица 16.4. Параметры xml2sql.pl

Параметр	Описание
`-sn servername`	Имя источника данных
`-driver dbi_driver`	Драйвер, используемый DBI (по умолчанию – ODBC)
`-uid username`	Имя пользователя
`-pwd password`	Пароль (необязательно)
`-table tablename`	Создаваемая таблица
`-input inputfile`	Файл, из которого будет считываться ввод XML
`-x`	Удалить содержимое таблицы перед вставкой

Для работы с ним необходимо установить модуль `XML::Parser`, опять-таки из CPAN.

Есть и другие пакеты, такие как XML-DBMS Рона Буррита (Ron Bourret), позволяющий осуществлять более сложное взаимодействие, но они выходят за рамки нашего описания. Дополнительная информация может быть найдена по адресу *http://www.rpbourret.com/xmldbms/*.

Резюме

Были исследованы различные пути доступа к базам данных PostgreSQL из Perl, мощные возможности Perl в обработке строк позволяют избежать написания кода на C, который реализовал бы то же самое. Хотя существует множество способов использовать базы данных совместно с Perl, рассмотрены были два, вероятно, самых важных для программирования PostgreSQL: прямой, практически копирующий интерфейс C, и не зависящий от базы данных уровень DBI.

В рамках DBI можно использовать разные серверы баз данных с одним и тем же клиентским кодом. Чтобы сделать программирование еще проще, существует ряд расширений кода DBI в виде модулей `DBIx`.

17

Доступ к PostgreSQL из Java

Неофициальным стандартом, которому следуют программы на языке Java, осуществляя доступ к реляционным базам данных, является JDBC. В этой главе подробно рассмотрен JDBC API и его применение Java-программами для доступа к реляционным данным, находящимся в базах PostgreSQL.

Итак, в данной главе обсуждаются:

- Общее представление о JDBC
- Драйверы JDBC
- Драйвер JDBC для PostgreSQL
- Соединение JDBC
- Операторы JDBC для выполнения разнообразных операторов DDL и DML
- Извлечение результирующих наборов из баз данных PostgreSQL
- Подготовленные операторы JDBC для выполнения предварительно скомпилированных запросов SQL
- Вызываемые операторы JDBC для выполнения хранимых процедур
- Групповые обновления
- Запросы метаданных базы данных и результирующего множества
- Краткий обзор новых возможностей API JDBC 3.0, окончательный проект которого в настоящее время предложен к обсуждению

Глава завершается созданием Java-приложения, имеющего графический пользовательский интерфейс и осуществляющего доступ к базе данных PostgreSQL.

Общее представление о JDBC

JDBC – это стандартный API-интерфейс, посредством которого Java-программы могут получать доступ к внешним менеджерам ресурсов, главным образом к реляционным базам данных, независящим от менеджера ресурсов способом. То есть Java-приложение, написанное с применением стандартных классов и интерфейсов JDBC, может переноситься на базы данных других производителей реляционных СУБД, если они поддерживают SQL, удовлетворяющий стандарту ANSI. JDBC API включает в себя базовый JDBC API и расширенный API.

Базовый API в основном определяет стандартные интерфейсы для:

- Создания соединения с базой данных
- Создания операторов
- Доступа к результирующим множествам
- Запроса метаданных базы данных и результирующих множеств

Базовые классы и интерфейсы определены в пакете `java.sql` и доступны в стандартной редакции Java 2 (Java 2 Platform, Standard Edition).

Расширенный API определяет более сложные интерфейсы для обработки XA-ресурсов, распределенных транзакций, пулов соединений и диспетчеров соединений. XA-ресурсы могут использоваться для обработки распределенных транзакций и двухфазных операций фиксации (two-phase commit), когда одна транзакция должна охватывать несколько баз данных. Такие классы и интерфейсы принадлежат пакету `javax.sql` и доступны в Java 2 Platform, Enterprise Edition.

В данной главе сосредоточимся на стандартном JDBC.

Драйверы JDBC

JDBC API лишь определяет интерфейсы для объектов, предназначенных для выполнения различных задач, связанных с базами данных: открытия и закрытия соединений, выполнения операторов SQL и извлечения результирующих множеств. Он не предоставляет классов реализации для этих интерфейсов. Переносимые программы на языке Java не должны знать о классах реализации, им следует использовать только стандартные интерфейсы.

Будучи верноподданными объектно-ориентированного программирования, все мы пишем программы для интерфейсов, а не реализаций. Для стандартных интерфейсов JDBC классы реализации предоставляются производителями менеджеров ресурсов или же третьими сторонами. Такие программные реализации называются **драйверами JDBC**. Драйверы JDBC преобразуют стандартные вызовы JDBC в вызовы API, присущего внешнему менеджеру ресурсов. Приведенная ниже

схема (рис. 17.1) показывает, как клиентское приложение базы данных, написанное на Java, обращается к внешнему менеджеру ресурсов при помощи прикладного интерфейса JDBC и драйвера:

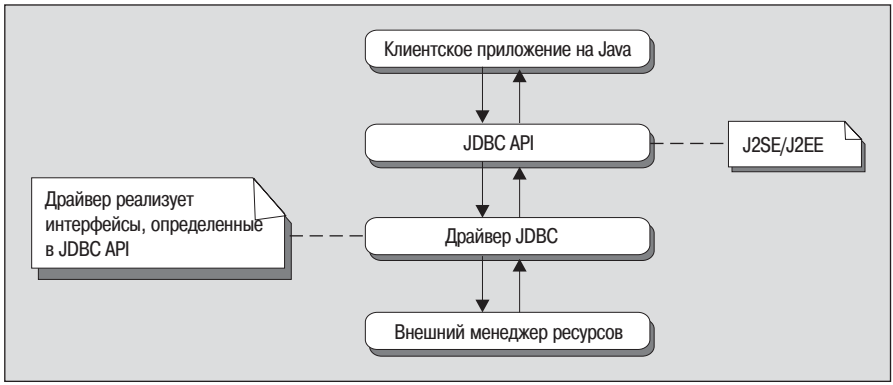


Рис. 17.1. Доступ клиентского приложения, написанного на Java, к внешним ресурсам

В зависимости от механизма реализации драйверы JDBC можно грубо разделить на четыре типа.

Тип 1

Драйверы JDBC типа 1 реализуют JDBC API поверх низкоуровневого программного интерфейса, такого как ODBC. Как правило, они непереносимы, поскольку зависят от внутренних библиотек. Эти драйверы транслируют вызовы JDBC в вызовы ODBC, а ODBC посылает запрос внешнему источнику данных, используя вызовы внутренних библиотек. Драйвер JDBC-ODBC, входящий в дистрибутив J2SE, является примером драйвера первого типа.

Тип 2

Драйверы типа 2 написаны на смеси Java и внутреннего кода. Такие драйверы используют специальные собственные API фирм-разработчиков для доступа к источнику данных. Они преобразуют вызовы JDBC в специальные вызовы производителей, вызывая внутренние библиотеки производителя. Эти драйверы также не переносимы (как и тип 1), поскольку зависят от внутреннего кода.

Тип 3

Драйверы типа 3 используют промежуточный посреднический сервер для доступа к внешним источникам данным. Обращения к промежуточному серверу не зависят от базы данных. Однако промежуточный

сервер выполняет специальные внутренние вызовы фирм-разработчиков для доступа к источникам данным. В этом случае драйвер целиком пишется на Java.

Тип 4

Драйверы типа 4 написаны на чистом Java, они реализуют интерфейсы JDBC и преобразуют специальные вызовы JDBC в характерные для фирмы-производителя обращения к данным. Они реализуют протокол передачи данных и сетевой протокол для целевого менеджера ресурсов. Большинство ведущих производителей баз данных предоставляют драйверы четвертого типа для доступа к их серверам баз данных.

Сборка драйвера JDBC PostgreSQL

Комплект программного обеспечения PostgreSQL включает в себя драйвер JDBC четвертого типа. В этом разделе поясняется, как скомпоновать драйвер JDBC для PostgreSQL из файлов исходных текстов. Этапы сборки драйвера JDBC для PostgreSQL перечислены ниже:

1. Загрузите и установите последнюю версию средства разработки `Ant` с сайта *http://jakarta.apache.org/builds/jakarta-ant/release/v1.3/bin/* для двоичного дистрибутива или *http://jakarta.apache.org/builds/jakarta-ant/release/v1.3/src/* для исходных текстов.
2. Распакуйте архив исходных текстов PostgreSQL в локальную файловую систему.
3. Внутри структуры каталога исходных текстов перейдите в `/src/interfaces/jdbc`.
4. Убедитесь, что в каталоге присутствует файл `build.xml`.
5. Запустите сценарий сборки `Ant`, введя `ant` в каталоге, указанном в шаге 3.
6. Убедитесь, что файлы всех необходимых классов Ant присутствуют в пути классов: Ant-классы, доступные в библиотечном каталоге установки Ant и соответствующий JAXP синтаксический анализатор XML. Примером соответствующего JAXP синтаксического анализатора XML является Xerces из Apache, его можно загрузить с сайта *http://xml.apache.org*. Если вы работаете с Xerces, проверьте наличие файла `xerces.jar` в пути классов.

В случае успешного завершения в каталоге вывода, определенном в файле `build.xml`, создаются два файла. Первый, `postgresql.jar`, содержит драйвер JDBC, а второй, `postgresql-examples.jar`, — примеры. Поскольку средство разработки Ant написано на Java, то указанные выше шаги можно выполнять как в среде Linux, так и в Cygwin.

Существует и другая возможность: скачайте заранее скомпилированные драйверы JDBC для PostgreSQL с сайта *http://jdbc.postgresql.org/download/*.

DriverManager и Driver

Пакет `java.sql` определяет интерфейс `java.sql.Driver`, который должен реализовываться всеми драйверами JDBC, и класс `java.sql.DriverManager`, работающий как интерфейс к клиентским приложениям базы данных для выполнения таких задач, как соединения с внешними менеджерами ресурсов и вывод отладочных сообщений. Когда клиент JDBC передает `DriverManager` запрос на установку соединения с внешним менеджером ресурсов, то он поручает задачу соответствующему классу драйверов, реализованному предоставленным драйвером JDBC или производителем менеджера ресурсов, или третьей стороной. В данном разделе подробно поговорим о ролях, которые `java.sql.DriverManager` и `java.sql.Driver` играют в JDBC API.

java.sql.DriverManager

Первоочередной задачей класса `DriverManager` является отслеживание имеющихся разнообразных драйверов JDBC. В разделе, посвященном `java.sql.Driver`, мы узнаем о том, как драйверы JDBC самостоятельно регистрируются в `DriverManager`. Этот класс также предоставляет методы для:

- Обеспечения соединения с базой данных
- Управления журналами JDBC
- Установки максимального времени входа в систему

Управление драйверами

В данном разделе обсуждаются методы, предусматриваемые классом `DriverManager` для управления драйверами:

```
public static void registerDriver(Driver driver) throws SQLException
```

Этот метод обычно применяется классами реализации интерфейса `java.sql.Driver`, предоставленными драйверами JDBC, для регистрации самих себя при помощи `DriverManager`. Он вызывает экземпляр `java.sql.SQLException` в случае ошибки базы данных. `DriverManager` использует зарегистрированные драйверы для делегирования запросов на соединение с базой данных.

```
public static void deregisterDriver(Driver driver) throws SQLException
```

Этот метод служит для отмены регистрации зарегистрированного в DriverManager драйвера.

```
public static Enumeration getDrivers()
```

Данный метод возвращает перечень всех зарегистрированных на настоящий момент драйверов DriverManager.

```
public static Driver getDriver(String url) throws SQLException
```

Метод применяется для получения зарегистрированного в DriverManager драйвера, соответствующего переданному JDBC URL. Он выдает исключение SQLException в случае ошибки доступа к базе данных. JDBC URL применяются для однозначной идентификации типа и местоположения менеджера ресурсов. Это означает, что хотя драйвер JDBC и может обработать любое количество соединений, отождествленных с различными JDBC URL, базовый формат URL, включающий в себя протокол и подпротокол, является специфичным для используемого драйвера.

Клиентские приложения JDBC указывают JDBC URL при запросе на соединение. DriverManager может найти драйвер, соответствующий заданному URL, в списке зарегистрированных драйверов и передать ему запрос на установление соединения, если соответствие найдено. Обычно JDBC URL имеет такой формат:

```
<protocol>:<sub-protocol>:<resource>
```

Протокол – это всегда jdbc, а подпротокол зависит от типа используемого менеджера ресурсов. Формат URL для PostgreSQL таков:

```
jdbc:postgres://<host>:<port>/<database>
```

Здесь host – это адрес сервера, на котором запущен postmaster, а database – имя базы данных, с которой хочет соединиться клиентское приложение.

Управление соединениями

В данном разделе поговорим о методах класса DriverManager, управляющих соединениями с базами данных.

```
public static Connection getConnection(String url) throws SQLException
```

Этот метод открывает соединение с базой данных, заданной JDBC URL. О классе java.sql.Connection подробно рассказывается в следующем разделе. Он выдает исключение SQLException в случае ошибки доступа к базе данных.

```
public static Connection getConnection(String url,String user,String
password) throws SQLException
```

Этот метод открывает соединение с базой данных, заданной JDBC URL, основываясь на указанном имени пользователя и пароле. Он выдает исключение SQLException в случае ошибки доступа к базе данных.

```
public static Connection getConnection(String url,Properties info) throws
SQLException
```

Этот метод открывает соединение с базой данных, заданной JDBC URL. Экземпляр класса java.util.Properties предназначен для указания конфиденциальных данных. Название свойства для задания пользователя – это user, а для пароля – password. Он генерирует исключение SQLException в случае ошибки доступа к базе данных.

Управление журналами JDBC

В этом разделе обсуждаются методы, предоставляемые классом DriverManager для управления журналами JDBC:

```
public static PrintWriter getLogWriter()
```

Данный метод извлекает дескриптор экземпляра класса java.io.PrintWriter, в который записывается информация трассировки и данные журнала регистрации работ. Он выдает исключение SQLException в случае ошибки доступа к базе данных:

```
public static void setLogWriter(PrintWriter writer)
```

Этот метод устанавливает PrintWriter, в который DriverManager записывает все события в процессе выполнения и все зарегистрированные драйверы:

```
public static void println(String message) throws SQLException
```

Метод пишет сообщение в текущий поток журнала регистрации событий.

Управление временными лимитами входа в систему

В этом разделе рассмотрены методы, предоставляемые классом DriverManager для управления входом в систему:

```
public static int getLoginTimeout()
```

Данный метод получает максимальное время (в секундах), в течение которого DriverManager будет ожидать установления соединения:

```
public static void setLogWriter(PrintWriter writer)[1]
```

[1] Похоже, что здесь ошибка. Должно быть нечто вроде setLogTimeout (int timeout). – *Примеч. науч. ред.*

Метод устанавливает максимальное время (в секундах), в течение которого DriverManager будет ожидать установления соединения.

java.sql.Driver

Этот интерфейс определяет методы, которые должны реализовываться всеми драйверами JDBC. Все классы реализации драйверов должны иметь статический код инициализации для регистрации в DriverManager, который ищет драйвер в списке зарегистрированных драйверов и передает запрос на соединение соответствующему классу драйверов в зависимости от указанного JDBC URL. Посмотрим на исходный текст класса org.postgresql.Driver, представляющий собой реализацию драйвера для PostgreSQL, рассматриваемую в данной главе (этот класс находится в каталоге interfaces/jdbc/org/postgresql внутри вашей структуры исходных текстов PostgreSQL). Вы найдете такой фрагмент кода:

```
static {
    try {
    java.sql.DriverManager.registerDriver(new Driver());
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Каждый раз, когда загрузчик классов загружает класс org.postgresql.Driver, он регистрируется в DriverManager. Следовательно, клиентские приложения JDBC должны загрузить определение класса драйвера, который они хотят использовать, чтобы DriverManager мог использовать его для установления соединений с базой данных. Очевидным путем осуществления этого является использование статического метода forName() для класса java.lang.Class, как показано ниже:

```
try {
    Class.forName("org.postgresql.Driver");
}catch(ClassNotFoundException e) {
    //Обработка исключения
}
```

Если класс org.postgresql.Driver не обнаружен в classpath, то возбуждается исключение ClassNotFoundException. Поэтому необходимо убедиться, что файл postgres.jar, содержащий необходимые классы, присутствует в пути классов.

Приведенная ниже диаграмма взаимодействия (рис. 17.2) изображает типичное клиентское приложение JDBC, устанавливающее соединение с базой данных PostgreSQL под названием bpsimple, работающей на локальной машине. Указаны имя пользователя «meeraj» и пароль «waheeda».

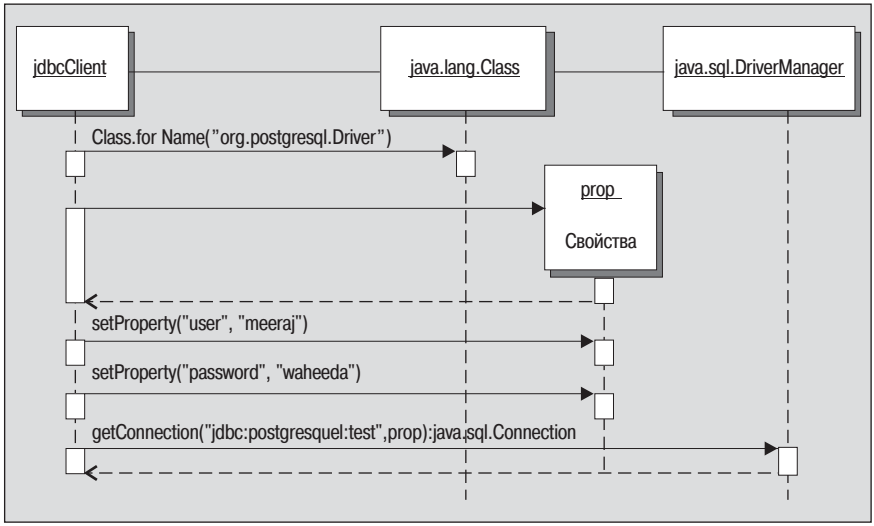


Рис. 17.2. Соединение клиента JDBC с базой данных PostgreSQL

Следующий фрагмент кода соответствует последовательности событий, представленной на схеме, приведенной выше:

```
try {

    //Загрузить драйвер JDBC
    Class.forName("org.postgresql.Driver");

    //создать объект properties с именем пользователя и паролем
    Properties prop = new Properties();
    prop.setProperty("user","meeraj");
    prop.setProperty("password","waheeda");

    //Задать JDBC URL
    String url = "jdbc:postgresql:test";

//Открыть соединение
    Connection con = DriverManager.getConnection(url,prop);

}catch(ClassNotFoundException e) {
    //Обработать исключение
}catch(SQLException e) {
    // Обработать исключение
}
```

Теперь посмотрим на методы, определенные интерфейсом `java.sql.Driver`:

```
public boolean acceptsURL(String url) throws SQLException
```

Этот метод возвращает значение «истина», если класс реализации драйвера может открыть соединение с указанным URL. Классы реализации обычно возвращают значение «истина», если они могут распознать подпротокол, заданный в JDBC URL. В случае ошибки доступа к базе данных возбуждается исключение `SQLException`.

```
public Connection connect(String url,Properties info) throws SQLException
```

Данный метод возвращает соединение с указанным URL, основываясь на свойствах, определенных в аргументе `info`. Обычно `DriverManager` вызывает этот метод, получая запрос на установление соединения от клиентских приложений JDBC. В случае ошибки доступа к базе данных возбуждается исключение `SQLException`.

```
public int getMajorVersion()
```

Этот метод возвращает основной номер версии драйвера.

```
public int getMinorVersion()
```

Этот метод возвращает дополнительный номер версии драйвера.

```
public boolean jdbcCompliant()
```

Данный метод возвращает значение «истина», если драйвер совместим с JDBC. Полностью совместимый с JDBC драйвер должен строго следовать требованиям JDBC API и SQL92 Entry Level. Разнообразные стандарты и спецификации, относящиеся к SQL, приведены по адресу *http://www.opengroup.org*. В случае несоответствия метод возвращает значение «ложь».

Соединения

Интерфейс `java.sql.Connection` определяет методы, необходимые для постоянного соединения с базой данных. Производитель драйверов JDBC реализует этот интерфейс. Нейтральные по отношению к производителям баз данных клиентские приложения никогда не используют класс реализации, а только интерфейс. Данный интерфейс определяет методы для следующих задач:

- Операторы, подготовленные операторы и вызываемые операторы – это разные типы операторов для выполнения SQL-запросов клиентских JDBC-приложений к базе данных. Операторы будут описаны в последующих разделах.
- Получение и установка режима автофиксации (auto-commit).
- Получение метаинформации о базе данных.
- Фиксация и откат транзакций.

В данном разделе подробно рассказано о различных методах, определенных в интерфейсе `java.sql.Connection`.

Создание объектов Statement

Интерфейс java.sql.Connection определяет ряд методов для создания операторов, применяемых для выполнения SQL-запросов к базе данных:

```
public Statement createStatement() throws SQLException
```

Этот метод предназначен для создания экземпляров интерфейса java.sql.Statement. Данный интерфейс может применяться для отправки запросов SQL к базе данных. Интерфейс java.sql.Statement обычно служит для выполнения простых запросов SQL, которые не принимают параметров. В случае ошибки доступа к базе данных возбуждается исключение SQLException:

```
public Statement createStatement(int resType, int resConcurrency) throws
SQLException
```

Все аналогично предыдущему методу, только клиентское приложение JDBC получает возможность указать тип результирующего множества и параллелизм доступа к нему. Результирующие множества служат для извлечения результатов из базы данных в клиентское приложение. Поговорим о них чуть позже в этой главе. Тип результирующего множества определяет направление обхода множества, а параллелизм доступа – способ одновременного обращения множественных потоков к результирующему множеству.

```
public PreparedStatement prepareStatement(String sql) throws SQLException
```

Этот метод предназначен для создания экземпляров интерфейса java.sql.PreparedStatement. Интерфейс java.sql.PreparedStatement обычно применяется для выполнения запросов SQL, принимающих любые аргументы. Подготовленные операторы (prepared statements) могут выполнять прекомпиляцию и сохранять операторы SQL. В случае ошибки доступа к базе данных возбуждается исключение SQLException. Операторы SQL, передаваемые подготовленному оператору, могут использовать заполнители параметров «?» для передачи входных (IN) параметров.

```
public Statement prepareStatement (String sql,int resType, int
resConcurrency) throws SQLException
```

Аналогичен предыдущему методу, только клиентское приложение JDBC получает возможность указать тип результирующего множества и параллелизм доступа к нему:

```
public CallableStatement prepareCall(String sql) throws SQLException
```

Такой метод служит для создания экземпляров интерфейса java.sql.CallableStatement. Интерфейс java.sql.CallableStatement обычно приме-

няется для вызовов хранимых процедур базы данных, принимающих параметры IN и OUT. В случае ошибки доступа к базе данных возбуждается исключение SQLException. В вызовах хранимых процедур, передаваемых подготовленным операторам, заполнители параметров «?» могут использоваться как для входных (IN), так и для выходных (OUT) параметров.

```
public Statement prepareCall (String sql,int resType, int resConcurrency)
throws SQLException
```

Аналогичен предыдущему методу, только клиентское приложение JDBC получает возможность указать тип результирующего множества и параллелизм доступа к нему.

Обработка транзакций

Интерфейс соединений определяет набор методов для обработки транзакций базы данных.

```
public void setAutoCommit(boolean autoCommit) throws SQLException
```

Этот метод устанавливает режим автоматической фиксации. Если ему передано значение «истина», то фиксация (commit) для операторов SQL выполняется автоматически, в противном случае клиентские приложения должны завершать транзакцию явным образом. В случае ошибки доступа к базе данных этот метод возбуждает исключение SQLException.

```
public boolean getAutoCommit() throws SQLException
```

Этот метод получает текущий режим автофиксации (auto-commit). В случае ошибки доступа к базе данных этот метод возбуждает исключение SQLException.

```
public void commit() throws SQLException
```

Данный метод фиксирует текущую транзакцию, сопоставленную соединению. В случае ошибки доступа к базе данных этот метод возбуждает исключение SQLException.

```
public void rollback() throws SQLException
```

Метод откатывает текущую транзакцию, сопоставленную соединению. В случае ошибки доступа к базе данных этот метод возбуждает исключение SQLException.

```
public int getTransactionIsolation() throws SQLException
```

Этот метод получает текущий уровень изоляции транзакций. Уровень изоляции транзакций, рассмотренный в главе 9, указывает, возможны ли неаккуратное считывание, неповторяемые считывания и фик-

тивные элементы. В случае ошибки доступа к базе данных этот метод возбуждает исключение SQLException.

```
public void getTransactionIsolation(int level) throws SQLException
```

Метод устанавливает уровень изоляции транзакций. В случае ошибки доступа к базе данных он возбуждает исключение SQLException.

Метаданные базы данных

Интерфейс соединения предоставляет метод для получения метаинформации базы данных:

```
public DatabaseMetaData getMetaData() throws SQLException
```

Этот метод возвращает экземпляр класса, реализующего интерфейс java.sql.DatabaseMetaData. В случае ошибки доступа к базе данных этот метод возбуждает исключение SQLException. Интерфейс java.sql.DatabaseMetaData предоставляет 149 методов получения информации о базе данных. Полный перечень методов можно найти в документации Javadocs для JDBC API.

Извлечение метаданных PostgreSQL

В этом разделе будет написан пример, выбирающий очень небольшое подмножество метаданных PostgreSQL. Класс сначала загружает драйвер JDBC для PostgreSQL и устанавливает соединение с базой данных bpsimple, запущенной на локальной машине localhost. Затем он получает дескриптор метаданных базы данных от соединения и выводит следующие сведения:

- Название СУБД
- Версия СУБД
- Основной номер версии драйвера
- Дополнительный номер версии драйвера
- Название драйвера
- Версия драйвера
- JDBC URL
- Поддержка транзакций
- Информация о том, использует ли PostgreSQL локальные файлы для хранения таблиц

Исходные тексты класса приведены ниже. Сохраните их в PostgreSQLMetaData.java:

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
```

```
public class PostgreSQLMetaData {

    public static void main(String args[]) throws Exception {

        Class.forName("org.postgresql.Driver");
        String url = "jdbc:postgresql:bpsimple";
        Connection con =
            DriverManager.getConnection(url,"meeraj","password");
        DatabaseMetaData dbmd = con.getMetaData();

        System.out.print("Database Product Name : ");
        System.out.println(dbmd.getDatabaseProductName());

        System.out.print("Database Product Version : ");
        System.out.println(dbmd.getDatabaseProductVersion());

        System.out.print("Driver Major Version : ");
        System.out.println(dbmd.getDriverMajorVersion());

        System.out.print("Driver Minor Version : ");
        System.out.println(dbmd.getDriverMinorVersion());

        System.out.print("Driver Name : ");
        System.out.println(dbmd.getDriverName());

        System.out.print("Driver Version : ");
        System.out.println(dbmd.getDriverVersion());

        System.out.print("JDBC URL : ");
        System.out.println(dbmd.getURL());

        System.out.print("Supports Transactions : ");
        System.out.println(dbmd.supportsTransactions());

        System.out.print("Uses Local Files : ");
        System.out.println(dbmd.usesLocalFiles());

        con.close();

    }

}
```

Скомпилируйте класс при помощи такой команды:

```
$ javac PostgreSQLMetaData.java
```

Вызовите интерпретатор Java для класса с файлом `postgresql.jar` в пути классов (classpath):

```
$ java PostgreSQLMetaData
```

В зависимости от версии PostgreSQL может быть получен следующий результат. При вызове JVM проверьте, присутствует ли файл `postgresql.jar` в пути классов системы:

```
Database Product Name : PostgreSQL
Database Product Version : 7.1
Driver Major Version : 7
Driver Minor Version : 1
Driver Name : PostgreSQL Native Driver
Driver Version : PostgreSQL 7.1 JDBC2
JDBC URL : jdbc:postgresql:test
Supports Transactions : true
Uses Local Files : false
```

Интерфейс метаданных базы данных может применяться для получения массы другой информации, например о названиях каталогов и таблиц и о поддерживаемых особенностях SQL. Подробная информация приведена в Javadoc.

Результирующие наборы данных JDBC

Набор данных JDBC представляет собой двумерный массив данных, порожденных в результате выполнения SQL-операторов `SELECT` с применением операторов JDBC. Об операторах JDBC подробно рассказано в следующем разделе. Результирующие наборы данных JDBC представлены интерфейсом `java.sql.ResultSet`. Производитель JDBC обеспечивает класс реализации для этого интерфейса.

Тип результирующего множества и параллелизм доступа к нему

Выполнение соответствующих методов для объектов `Statement` приводит к созданию объектов типа `java.sql.ResultSet`. В разделе, посвященном соединениям, говорилось, что при создании объектов-операторов можно указать тип и параллелизм доступа к результирующим множествам, которые могут быть созданы такими объектами-операторами. В данном разделе подробно рассмотрены режимы прокрутки и чувствительность результирующих множеств к изменениям данных.

Тип

Наборы данных могут принадлежать к одному из типов:

- `TYPE_FORWARD_ONLY`

  «Только вперед». Обход результирующих множеств выполняется только в прямом направлении. Другими словами, если текущий указатель курсора указывает на строку с номером N, вернуться обратно к строке с номером N–1 нельзя.

- TYPE_SCROLL_INSENSITIVE

  Этот тип объекта ResultSet допускает прокрутку и не воспринимает изменения, сделанные другими потоками.

- TYPE_SCROLL_SENSITIVE

  Этот тип объекта ResultSet допускает прокрутку и чувствителен к изменениям, сделанным другими потоками.

Интерфейс определяет метод получения типа результирующего множества. Установлен же тип может быть только при создании операторов с применением объектов-соединений (что объяснялось в предыдущем разделе):

```
public int getType() throws SQLException
```

Параллелизм доступа

Параллельный доступ к результирующим множествам может принадлежать к одному из двух типов:

- CONCUR_READ_ONLY

  Конкурентный доступ только для чтения, наборы результатов не обновляются.

- CONCUR _UPDATEABLE

  Наборы результатов обновляемы и доступны как для чтения, так и для записи.

Интерфейс ResultSet определяет метод для получения типа параллельного доступа к результирующему множеству. Установлен же он может быть только при создании операторов с использованием объектов-соединений (что объяснялось в предыдущем разделе):

```
public int getConcurrency() throws SQLException
```

Обход результирующих множеств

Интерфейс ResultSet определяет различные методы обхода результирующих множеств, управления положением курсора и направлением выборки. Эти методы мы и обсудим подробно в данном разделе.

Передвижение по результирующим множествам

Ниже рассматриваются методы передвижения по результирующим множествам.

```
public boolean next() throws SQLException
```

Этот метод перемещает указатель текущего курсора на следующую строку и возвращает значение «истина», если есть еще строки, в про-

тивном случае возвращается «ложь». В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public boolean first() throws SQLException
```

Метод перемещает указатель текущего курсора на первую строку и возвращает значение «истина», если курсор указывает на первую строку, в противном случае возвращается «ложь». Этот метод не может использоваться для курсоров, которым разрешено только прямое направление обхода. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public boolean last() throws SQLException
```

Данный метод перемещает указатель текущего курсора на последнюю строку и возвращает значение «истина», если курсор указывает на последнюю строку, в противном случае возвращая «ложь». Этот метод не может применяться для курсоров, которым разрешено только прямое направление обхода. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public boolean absolute(int rows) throws SQLException
```

Этот метод перемещает указатель текущего курсора вперед или назад от начала или конца результирующего множества на строку, указанную аргументом. Курсор перемещается вперед от начала, если значение rows положительно, и назад от конца множества, если значение rows отрицательно. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public boolean relative(int row) throws SQLException
```

Данный метод перемещает указатель текущего курсора вперед или назад с текущей позиции на строку, указанную аргументом. Курсор перемещается вперед, если значение rows положительно, и назад, если значение rows отрицательно. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public boolean previous() throws SQLException
```

Этот метод перемещает указатель текущего курсора на предыдущую строку. Он не может применяться для курсоров, которым разрешено только прямое направление обхода. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

Определение позиции курсора

Ниже рассмотрены различные методы запроса позиции курсора.

```
public boolean next() throws SQLException
```

Этот метод перемещает указатель текущего курсора на следующую строку и возвращает значение «истина», если есть еще строки, в противном случае возвращается «ложь». В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public boolean isBeforeFirst() throws SQLException
```

Возвращает «истину», если указатель курсора находится перед первой строкой. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public boolean isAfterLast() throws SQLException
```

Возвращает «истину», если указатель курсора находится за последней строкой. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public boolean isFirst() throws SQLException
```

Возвращает «истину», если указатель курсора находится на первой строке. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public void isLast() throws SQLException
```

Возвращает «истину», если указатель курсора находится на последней строке. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public void beforeFirst() throws SQLException
```

Перемещает курсор в начало результирующего множества, в позицию перед первой строкой. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public boolean afterLast() throws SQLException
```

Перемещает курсор в конец результирующего множества, в позицию за последней строкой. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

Направление и размер выборки

Теперь обратимся к методам управления размером и направлением выборки. Эти методы подсказывают драйверу направление, в котором будут выбираться строки и размер выборки, чтобы он мог соответственно выбирать записи из базы данных.

```
public int getFetchDirection() throws SQLException
```

Возвращает текущее направление выборки. В случае ошибки доступа к базе данных возбуждается исключение SQLException. JDBC API определяет три направления выборки:

- FETCH_FORWARD
- FETCH_REVERSE
- FETCH_UNKNOWN

```
public void setFetchDirection(int direction) throws SQLException
```

Этот метод устанавливает направление выборки. В случае ошибки доступа к базе данных, или если направление отличается от FETCH_FORWARD для результирующего множества, которому разрешен только прямой обход, возбуждается исключение SQLException.

```
public int getFetchSize() throws SQLException
```

Получает текущий размер выборки. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public void setFetchDirection(int direction) throws SQLException
```

Этот метод указывает драйверу размер выборки. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

Доступ к данным результирующих множеств

Интерфейс определяет методы для извлечения данных из текущей строки результирующего множества. Данные могут быть извлечены в виде соответствующих типов данных. Общий формат этих методов выглядит так: getXXX(int col), где XXX может принадлежать одному из типов int, short, string, а col – это номер столбца текущей строки, из которого выбираются данные. Нумерация столбцов начинается с 1.

Можно указывать и названия столбцов. Вне зависимости от типа данных все столбцы могут быть извлечены в виде символьной строки. Все эти методы возбуждают исключение SQLException в случае ошибки доступа к базе данных.

Некоторые из методов доступа к данным приведены в табл. 17.1, назначение же оставшихся нетрудно понять на основании приведенных здесь описаний после обращения к Javadoc:

Таблица 17.1. Методы извлечения данных результирующих множеств

Название метода	Назначение
public boolean getBoolean(int i) public boolean getBoolean(String col)	Получить данные указанного столбца как логический тип
public int getInt(int i) public inr getInt(String col)	Получить данные указанного столбца как целый тип

Название метода	Назначение
`public String getString(int i)` `public String getString(String col)`	Получить данные указанного столбца как символьную строку

Соответствие типов данных

В этом разделе устанавливается соответствие типов Java типам данных PostgreSQL и JDBC.

Отображение типов Java в типы данных PostgreSQL и JDBC представлено в табл. 17.2. Различные типы JDBC определены в классе `java.sql.Types`:

Таблица 17.2. Соотношение типов Java, JDBC и PostgreSQL

Тип Java	Тип JDBC	Тип PostgreSQL
`java.lang.Boolean`	`TINYINT`	`INT2`
`java.lang.Byte`	`TINYINT`	`INT2`
`java.lang.Short`	`SMALLINT`	`INT2`
`java.lang.Integer`	`INTEGER`	`INT4`
`java.lang.Long`	`BIGINT`	`INT8`
`java.lang.Float`	`FLOAT`	`FLOAT(7)`
`java.lang.Double`	`DOUBLE`	`FLOAT8`
`java.lang.Character`	`CHAR`	`CHAR(1)`
`java.lang.String`	`VARCHAR`	`TEXT`
`java.sql.Date`	`DATE`	`DATE`
`java.sql.Time`	`TIME`	`TIME`
`java.sql.Timestamp`	`TIMESTAMP`	`TIMESTAMP`
`java.lang.Object`	`JAVA_OBJECT`	`OID`

Обновляемые результирующие множества

Обновляемые результирующие множества могут быть получены при помощи операторов, созданных с указанием типа параллельного доступа `CONCUR_UPDATEABLE`. Данные обновляемых результирующих множеств можно модифицировать, добавляя и удаляяя строки. В этом разделе рассмотрим методы изменения состояния результирующих множеств.

Удаление данных

Интерфейс определяет метод для удаления текущей строки как из результирующего множества, так и из базы данных.

```
public void deleteRow() throws SQLException
```

Этот метод удаляет текущую строку из результирующего множества и из базы данных. Он не может быть вызван, если курсор указывает

на строку INSERT. О строках INSERT рассказано в разделе, посвященном вставке данных в базу данных. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public boolean rowDeleted() throws SQLException
```

Данный метод проверяет, была ли удалена текущая строка. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

Обновление данных

Интерфейс результирующих множеств определяет ряд методов updateXXX() для обновления данных в текущей строке набора результатов. Однако эти методы не обновляют лежащие в их основе данные из базы данных. Некоторые методы описаны в табл. 17.3, а назначение остальных можно выяснить в Javadoc.

Таблица 17.3. Методы обновления данных результирующих множеств

Название метода	Назначение
`public void updateBoolean(int i, boolean x)` `public void updateBoolean(String col, boolean x)`	Устанавливает данные заданного столбца в указанное логическое значение
`public void updateInt(int i, int x)` `public void updateInt(String col, int x)`	Устанавливает данные заданного столбца в указанное целое значение
`public void updateString(int i, String x)` `public void updateString(String col, String x)`	Устанавливает данные заданного столбца в указанное строковое значение

```
public void updateRow() throws SQLException
```

Этот метод может быть вызван для обновления базы данных, на основе которой построено результирующее множество, данными, измененными при помощи методов updateXXX(). В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public void refreshRow() throws SQLException
```

Метод обновляет текущую строку самыми свежими данными базы данных. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public void cancelRowUpdates() throws SQLException
```

Данный метод отменяет обновление, примененное к текущей строке. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public boolean rowUpdated() throws SQLException
```

Этот метод проверяет, была ли обновлена текущая строка. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

Вставка данных

В результирующих множествах есть специальная строка INSERT для добавления данных в основную базу данных. Можно переместить курсор на строку INSERT посредством следующего метода.

```
public boolean moveToInsertRow() throws SQLException
```

Курсор перемещается со строки INSERT обратно на предыдущую строку при помощи такого метода:

```
public boolean moveToCurrentRow() throws SQLException
```

Этот метод в случае ошибки доступа к базе данных или если результирующее множество является необновляемым, возбуждает исключение SQLException.

Когда курсор указывает на строку вставки, можно вызывать подходящие методы updateXXX() для задания данных для новой строки. Затем данный метод может быть вызван для создания новой записи в базе данных:

```
public boolean insertRow() throws SQLException
```

В случае ошибки доступа к базе данных или если курсор указывает не на строку INSERT, он возбуждает исключение SQLException.

Другие важные методы

В этом разделе представлены другие важные методы интерфейса java.sql.ResultSet:

```
public void close() throws SQLException
```

Метод освобождает ресурсы базы данных и JDBC. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public ResultSetMetaData getMetaData() throws SQLException
```

Этот метод получает метаданные результирующего множества как экземпляр класса реализации интерфейса java.sql.ResultSetMetaData. Данный интерфейс определяет совокупность методов для доступа к метаданным результирующего множества, в том числе:

- Имя каталога
- Имя класса столбца
- Счетчик столбца
- Размер отображения столбца

- Метку столбца
- Тип столбца
- Имя типа столбца

Полный перечень приведен в документации Javadoc. В случае ошибки доступа к базе данных возбуждается исключение `SQLException`.

Операторы JDBC

JDBC API определяет три типа операторов для отправки запросов SQL базе данных:

- **Операторы**

  Операторы (statements) обычно применяются для отправки простых запросов SQL без параметров. Необходимые методы для объектов `Statement` определяются интерфейсом `java.sql.Statement`. Поставщик драйвера JDBC предоставляет класс реализации для данного интерфейса.

- **Подготовленные операторы**

  Подготовленные операторы (prepared statements) обычно используются для прекомпилированных операторов SQL, принимающих входные параметры (`IN`). Необходимые методы для объектов `PreparedStatement` определяются интерфейсом `java.sql.PreparedStatement`. Этот интерфейс расширяет интерфейс `java.sql.Statement`.

- **Вызываемые операторы**

  Вызываемые операторы обычно служат для вызова хранимых процедур базы данных и могут принимать как входные (`IN`), так и выходные (`OUT`) аргументы. Методы, необходимые для объектов подготовленных операторов, определяются интерфейсом `java.sql.CallableStatement`. Этот интерфейс расширяет интерфейс `java.sql.PreparedStatement`.

  Вызываемые операторы не поддерживаются в данной редакции драйвера JDBC для PostgreSQL.

Объект Statement

Интерфейс `java.sql.Statement` как правило используется для выполнения запросов SQL без входных и выходных параметров. Поставщик драйвера JDBC предоставляет класс реализации для данного интерфейса. Общие методы, необходимые различным операторам JDBC, определены в этом интерфейсе, их можно приблизительно разделить на следующие категории:

- Выполнение операторов SQL
- Запрос результатов и результирующих множеств

- Обработка групп операторов SQL
- Другие методы

Интерфейс `java.sql.Statement` определяет методы для выполнения таких операторов SQL, как `SELECT`, `UPDATE`, `INSERT`, `DELETE` и `CREATE`.

```
public ResultSet executeQuery(String sql) throws SQLException
```

Этот метод может применяться для отправки в базу данных запроса оператора `SELECT` и получения обратно результата. Экземпляр `SQLException` вызывается в случае ошибки базы данных. Пример кода приведен ниже:

```
try {
    Connection con = DriverManager.getConnection(url,prop);
    Statement stmt = con.createStatement();
    ResultSet res = stmt.executeQuery("SELECT * FROM MyTable");
}catch(SQLException e) {
    //Обработка исключений
}
```

Этот код просто возвращает результирующее множество, содержащее все данные таблицы `MyTable`.

```
public boolean execute(String sql) throws SQLException
```

Данный метод может использоваться как хранимая процедура для выполнения оператора SQL, способного возвращать более одного набора данных. Метод возвращает значение «истина», если следующий результат является объектом `ResultSet`. В случае ошибки доступа к базе данных возбуждается исключение `SQLException`.

```
public int executeUpdate(String sql) throws SQLException
```

Данный метод может применяться для выполнения операторов SQL, которые не возвращают результирующих множеств (это такие операторы, как `INSERT`, `UPDATE` и `DELETE`), а также операторов DDL. Он возвращает количество строк, которые были модифицированы. В случае ошибки доступа к базе данных возбуждается исключение `SQLException`.

Запрос результатов и результирующих множеств

Интерфейс `Statement` определяет разнообразные методы получения информации о результате выполнения оператора SQL:

```
public ResultSet getResultSet() throws SQLException
```

Несмотря на то что выполнение оператора SQL может привести к созданию нескольких результирующих множеств, в каждый момент времени только одно результирующее множество может быть открыто для объекта `Statement`. Этот метод возвращает текущее результирую-

щее множество, сопоставленное объекту Statement. Метод возвращает NULL, если больше нет доступных результирующих множеств или если следующим результатом является счетчик обновлений, генерируемый при выполнении оператора UPDATE, INSERT или DELETE. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public int getUpdateCount() throws SQLException
```

Данный метод возвращает счетчик обновлений для последнего выполненного оператора UPDATE, INSERT или DELETE. Метод возвращает -1 , если больше нет доступных счетчиков обновлений или если следующий результат представляет собой результирующее множество, сгенерированное при выполнении оператора SELECT. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public boolean getMoreResults() throws SQLException
```

Этот метод извлекает следующее результирующее множество для объекта Statement. Он возвращает «ложь», если больше нет доступных результирующих множеств или если следующим результатом является счетчик обновлений. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

Также существуют методы выполнения операций get (получения) и set (установки) для:

- Типа параллельного доступа к результирующему множеству, с которым был создан оператор
- Направления выборки результирующего множества
- Размера выборки

Обработка групп SQL

Интерфейс оператора также предоставляет методы для выполнения над базой данных группы операторов SQL:

```
public void addBatch(String sql) throws SQLException
```

Данный метод добавляет указанный sql в текущую группу. Обычно это операторы SQL UPDATE, INSERT или DELETE. Экземпляр SQLException вызывается в случае ошибки базы данных.

```
public void clearBatch() throws SQLException
```

Очищает текущую группу. В случае ошибки доступа к базе данных возбуждается исключение SQLException.

```
public int[] executeBatch() throws SQLException
```

Выполняет текущую группу. Метод возвращает массив обновленных счетчиков. В случае ошибки доступа к базе данных возбуждается исключение `SQLException`.

Другие методы

В эту группу входят методы:

- Получения и установки тайм-аута для запроса
- Закрытия оператора для освобождения ресурсов
- Получения и установки обработки escape-последовательностей
- Получения и очистки предупреждений SQL
- Получения и установки названий курсоров

Пример клиентского приложения JDBC

В этом разделе используем все полученные знания о JDBC. Напишем клиентское приложение JDBC, которое будет реализовывать следующие задачи:

- Установление соединения с базой данных
- Создание объекта `Statement`
- Вставка двух записей в таблицу клиентов
- Выборка этих двух записей из базы
- Удаление этих записей
- Закрытие соединения

Импортируем нужные классы:

```
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.DriverManager;

public class StatementClient {

    public static void main(String args[]) throws Exception {
```

Загрузим драйвер JDBC и установим соединение. Создадим объект `Statement`, используя соединение:

```
        Class.forName("org.postgresql.Driver");
        String url = "jdbc:postgresql:bpfinal";
        Connection con =
            DriverManager.getConnection(url,"meeraj","password");

        Statement stmt = con.createStatement();
```

Добавим два оператора SQL (для вставки записей в таблицу `customer`) в группу:

```
            System.out.println("Inserting records");
            stmt.addBatch("INSERT INTO customer(title,fname," +
                "lname,addressline,town,zipcode,phone) values " +
                "('Mr','Fred','Flintstone','31 Bramble Avenue'," +
                "'London','NT2 1AQ','023 9876')");
            stmt.addBatch("INSERT INTO customer(title,fname," +
                "lname,addressline,town,zipcode,phone) values " +
                "('Mr','Barney','Rubble','22 Ramsons Avenue'," +
                "'London','PWD LS1','111 2313')");
```

Выполним группу:

```
            stmt.executeBatch();
            System.out.println("Records Inserted");
            System.out.println();
```

Выберем записи из таблицы `customer` и выведем содержимое на стандартный вывод:

```
            System.out.println("Selecting records");
            String selectSQL = "SELECT * FROM customer";
            ResultSet res = stmt.executeQuery(selectSQL);

            while(res.next()) {
            for(int i = 1;i <= res.getMetaData().getColumnCount();i++) {
                System.out.print(res.getString(i) + "\t");
            }
            System.out.println();
            }
            System.out.println();
```

Удалим записи из таблицы `customer` и выведем количество удаленных записей:

```
            System.out.println("Deleting records");
            String deleteSQL = "DELETE FROM customer";
            System.out.println("Records deleted: " +
                stmt.executeUpdate(deleteSQL));
```

Закроем результирующее множество и соединение, чтобы освободить ресурсы:

```
            res.close();
            stmt.close();
            con.close();

        }

    }
```

Назовем класс StatementClient.java, скомпилируем класс и запустим для скомпилированного класса JVM (в пути классов должен быть файл postgresql.jar):

```
# java -cp ./postgresql.jar StatementClient
```

В зависимости от записей таблицы customer будет порожден вывод, подобный приведенному ниже. При вызове JVM убедитесь, что в переменной classpath вашей системы указан файл postgresql.jar:

```
Inserting records
Records Inserted

Selecting records
81   Mr Fred Flinstone  31 Bramble Avenue  London  NT2 1AQ  023 9876
82   Mr Barney Rubble  22 Ramsons Avenue  London  PWD LS1  111 2313

Deleting records
Records deleted: 2
```

Объекты PreparedStatements

Подготовленные операторы (prepared statements) применяются для выполнения предварительно скомпилированных операторов SQL и создаются в JDBC API при помощи интерфейса java.sql.PreparedStatement. Этот интерфейс расширяет интерфейс java.sql.Statement. Класс реализации для данного интерфейса предоставляет производитель драйверов JDBC. Подготовленные операторы создаются при помощи объектов соединений, как было показано ранее. Они также служат для выполнения операторов SQL с заменителями входных параметров, заданными при помощи символа «?». Подготовленные операторы рекомендуется применять для многократного выполнения операторов SQL с разными значениями параметров IN.

Методы, определенные в java.sql.PreparedStatement, можно условно разбить на следующие виды (вдобавок к уже определенным в интерфейсе операторов):

- Методы для выполнения операторов SQL
- Методы для обработки групп SQL
- Методы для установки значений входных (IN) параметров SQL, если в операторе SQL использованы заполнители

Выполнение операторов SQL

Интерфейс java.sql.PreparedStatement определяет методы для выполнения различных операторов SQL, таких как SELECT, UPDATE, INSERT, DELETE и CREATE. В отличие от соответствующих методов, определенных в интерфейсе операторов, эти методы не принимают операторы SQL как

аргументы. Операторы SQL определяются, когда подготовленные операторы создаются при помощи объектов соединений:

```
public ResultSet executeQuery() throws SQLException
```

Этот метод может применяться для выполнения оператора выборки, сопоставленного подготовленному оператору, и извлечения результата. В случае ошибки доступа к базе данных возбуждается исключение `SQLException`. Пример приведен ниже:

```
try {
    String sql = "SELECT * FROM customer WHERE fname = ? ";
    Connection con = DriverManager.getConnection(url,prop);
    PreparedStatement stmt = con.prepareStatement(sql);
    stmt.setString(1, "Fred");
    ResultSet res = stmt.executeQuery();
}catch(SQLException e) {
    //Handle exception
}
```

```
public boolean execute() throws SQLException
```

Этот метод может применяться для выполнения оператора SQL, соответствующего подготовленному оператору. Он возвращает значение «истина», если следующим результатом является объект `ResultSet`. При ошибке доступа к базе данных возбуждается исключение `SQLException`.

```
public int executeUpdate() throws SQLException
```

Метод может применяться для выполнения оператора SQL, соответствующего подготовленному оператору, который не возвращает результирующего множества, как операторы вставки, обновления и т. д. Метод возвращает количество строк, модифицированных оператором SQL. В случае ошибки доступа к базе данных возбуждается исключение `SQLException`.

Обновление данных

Интерфейс подготовленных операторов определяет набор методов `setXXX()` для установки значений параметров `IN` прекомпилированного оператора SQL, на месте которых стоят символы «`?`». Параметры индексируются начиная с `1`. Применяемый метод `setXXX()` должен быть совместим с ожидаемым типом SQL. Некоторые из этих методов перечислены в табл. 17.4, а назначение остальных нетрудно понять на основании этого описания при помощи документации Javadoc:

Таблица 17.4. Методы установки значений входных параметров

Название	Назначение
`public void setBoolean(int index, boolean x)`	Устанавливает входной (`IN`) параметр, заданный аргументом `index`, в логическое значение, определенное в `x`

Название	Назначение
`Public void setInt(int index, int x)`	Устанавливает входной (`IN`) параметр, заданный аргументом `index`, в целое значение, определенное в `x`
`public void setString(int index, string x)`	Устанавливает входной (`IN`) параметр, заданный аргументом `index` в строковое значение, определенное в `x`

Интерфейс также определяет метод для очистки текущих значений всех параметров сразу:

```
public void clearParameters() throws SQLException
```

Пример использования подготовленных операторов

Теперь перепишем предыдущий пример, используя подготовленные операторы, и посмотрим, как один и тот же оператор `INSERT` может выполняться несколько раз для разных значений:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.DriverManager;

public class PreparedStatementClient {

    public static void main(String args[]) throws Exception {

        Class.forName("org.postgresql.Driver");
        String url = "jdbc:postgresql:bpfinal";
        Connection con =
            DriverManager.getConnection(url,"meeraj","password");

        PreparedStatement stmt;

        String insertSQL = "INSERT INTO customer(title,fname," +
            "lname,addressline,town,zipcode,phone) VALUES " +
            "(?,?,?,?,?,?,?)";

        stmt = con.prepareStatement(insertSQL);

        System.out.println("Inserting records");

        stmt.setString(1,"Mr");
        stmt.setString(2,"Fred");
        stmt.setString(3,"Flinstone");
        stmt.setString(4,"31 Bramble Avenue");
        stmt.setString(5,"London");
        stmt.setString(6,"NT2 1AQ");
        stmt.setString(7,"023 9876");
        stmt.executeUpdate();
```

```
            stmt.clearParameters();

            stmt.setString(1,"Mr");
            stmt.setString(2,"Barney");
            stmt.setString(3,"Rubble");
            stmt.setString(4,"22 Ramsons Avenue");
            stmt.setString(5,"London");
            stmt.setString(6,"PWD LS1");
            stmt.setString(7,"111 2313");
            stmt.executeUpdate();

            System.out.println("Records Inserted");
            System.out.println();

            System.out.println("Selecting records");
            String selectSQL = "SELECT * FROM customer";
            stmt = con.prepareStatement(selectSQL);
            ResultSet res = stmt.executeQuery();
            while(res.next()) {
            for(int i = 1;i <= res.getMetaData().getColumnCount();i++) {
                System.out.print(res.getString(i) + "\t");
            }
            System.out.println();
            }
            System.out.println();

            System.out.println("Deleting records");
            String deleteSQL = "DELETE FROM customer";
            stmt = con.prepareStatement(deleteSQL);
            System.out.println("Records deleted: " +
                stmt.executeUpdate());

            res.close();

            stmt.close();
            con.close();

        }

    }
```

Исключительные ситуации и предупреждения SQL

Базовый JDBC API предусматривает четыре исключительных ситуации:

- `BatchUpdateException`

  Такое исключение порождается, если ошибка происходит при выполнении группы операторов SQL. Этот класс предлагает метод для

получения количества обновлений, которые были успешно выполнены в группе, в виде массива целых чисел.

- `DataTruncation`

  Такое исключение порождается, когда данные непредвиденно усекаются при чтении или записи. Этот класс предоставляет методы для доступа к следующей информации:

  - Количество байт, подлежащих передаче
  - Количество реально переданных байт
  - Произошло ли усечение для столбца или же параметра
  - Произошло ли усечение при чтении или же записи
  - Индекс столбца или параметр

- `SQLException`

  Это суперкласс всех других исключений SQL. Данный класс предоставляет методы доступа к коду ошибки базы данных и к статусу SQL для ошибки, породившей это исключение.

- `SQLWarning`

  Этот подкласс `SQLException`, указывающий на предупреждения во время доступа к базе данных.

Приложение JDBC с графическим интерфейсом пользователя

В этом разделе мы займемся разработкой небольшого приложения JDBC с графическим интерфейсом пользователя для ведения данных таблицы `customer`. Были сформулированы требования к приложению, и на их основе была сконструирована схема вариантов использования, представленная на рис. 17.3:

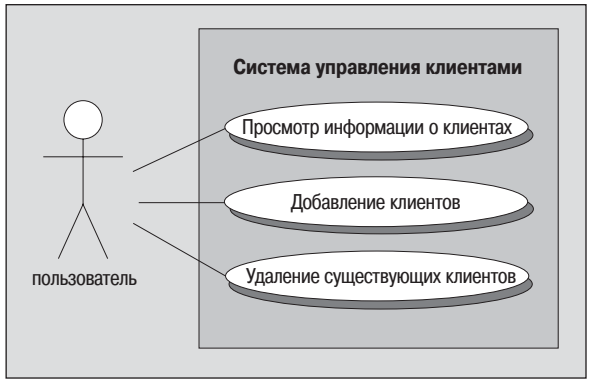


Рис. 17.3. Диаграмма вариантов использования для создаваемого приложения

Определены следующие варианты использования:

- Просмотр информации о клиентах в списке
- Добавление новых клиентов
- Удаление существующих клиентов

Реализуем приложение как автономное приложение Java с применением классов библиотеки Swing. Ввод информации о существующих клиентах выполняется посредством доступного только для чтения списка, вероятно, реализованного при помощи `JTable`, и предоставлять форму для ввода сведений о новом клиенте с помощью `JPanel` с необходимыми полями для ввода информации.

Схема классов

После тщательного анализа и планирования для моделирования системы были выбраны следующие классы:

- Основной класс для запуска приложения
- Класс для создания сущности «клиент»
- Класс, реализующий интерфейс `TableModel`, который будет действовать как модель данных для списка, отображающего сведения о клиентах
- Класс для создания формы ввода информации о новом клиенте

Customer

Этот класс создает в базе данных сущность «клиент»:

- Закрытые (private) переменные экземпляра, соответствующие столбцам таблицы `customer`
- Методы получения (accessors) и изменения (mutators) для всех атрибутов
- Конструктор для инициализации значений переменных экземпляра

CustomerTableModel

Данный класс действует как модель для экземпляра `JTable`, отображающего клиентскую информацию. Класс реализует интерфейс `javax.swing.table.TableModel`, расширяя класс `javax.swing.table.AbstractTableModel`:

- Этот класс хранит коллекцию экземпляров класса `Customer`, представляющую клиентов, информация о которых в настоящее время хранится в базе данных

- Он реализует методы обратных вызовов, необходимые контроллеру для отображения данных. Обратите внимание, что `JTable` и `TableModel` вписываются в классический шаблон MVC
- Он предоставляет метод для задания текущего списка клиентов
- Также предоставляется метод для удаления клиента из списка по заданному индексу

CustomerApp

Данный класс моделирует форму, используемую для ввода информации о новом клиенте:

- Предусмотрены поля ввода для таких сведений о клиенте, как обращение («титул»), имя, фамилия, адрес и телефон
- Предоставляется метод, возвращающий объект `Customer`, созданный из значений, введенных пользователем
- Необходимые поля инициализируются и помещаются в ячейки конструктором
- Также предоставляется метод для очистки текущих значений формы

CustomerPanel

Этот класс содержит основной метод приложения:

- Класс имеет в своем распоряжении переменные экземпляра типа `CustomerPanel` и `CustomerTableModel`
- Класс расширяет `javax.swing.JFrame`
- При запуске приложения устанавливается соединение с базой данных, извлекаются данные из таблицы `customer` и заполняется экземпляр класса `CustomerTableModel`
- Этот объект служит для инициализации экземпляра `JTable`, который отображает клиентскую информацию и добавляется во фрейм
- Экземпляр класса `CustomerPanel` также добавляется во фрейм
- Две кнопки, одна для добавления нового клиента, а другая – для удаления клиента, выбранного в списке, добавляются во фрейм.
- Класс предоставляет методы обратных вызовов для обработки событий при нажатии кнопок и закрытии фрейма. Когда фрейм закрыт, соединение с базой данных закрывается.

Схема классов для системы изображена на рис. 17.4:

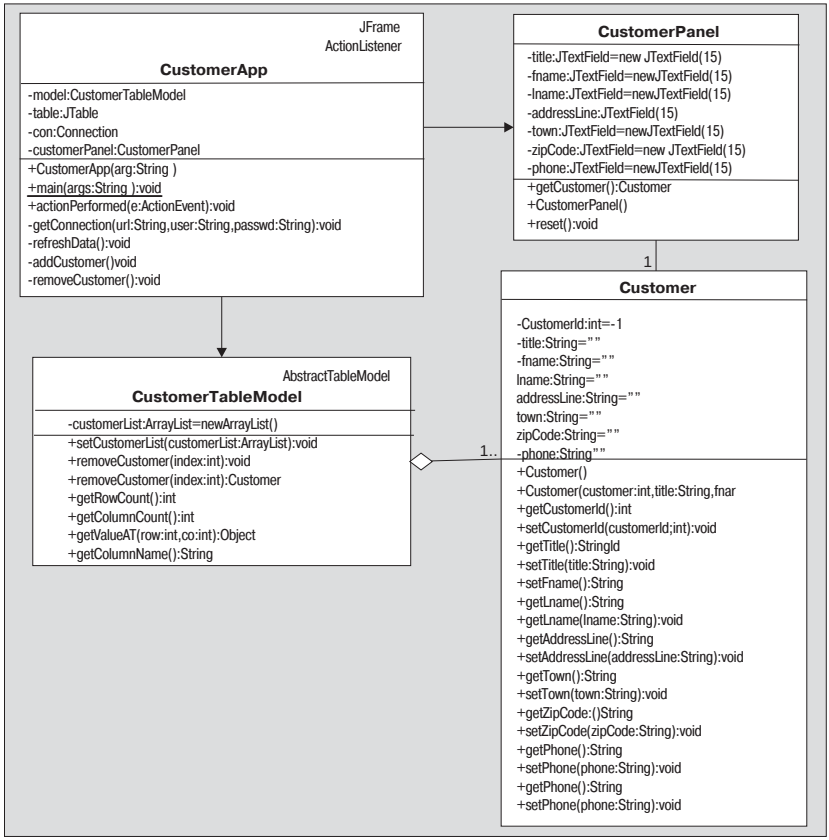


Рис. 17.4. Схема классов системы

Взаимодействие с системой

В данном разделе рассмотрена последовательность событий, вызванных случаями использования, определенными для системы.

Просмотр информации о клиентах

Приведенная диаграмма взаимодействия (рис. 17.5) изображает последовательность событий, относящихся к просмотру сведений о клиентах:

- Основной класс устанавливает соединение с базой данных
- Создается экземпляр класса `CustomerTableModel`
- Теперь вызывается закрытый (private) метод `refreshData`
- Соединение используется для создания объекта `Statement`
- Объект `Statement` выполняет оператор SQL для получения всех записей таблицы `customer`

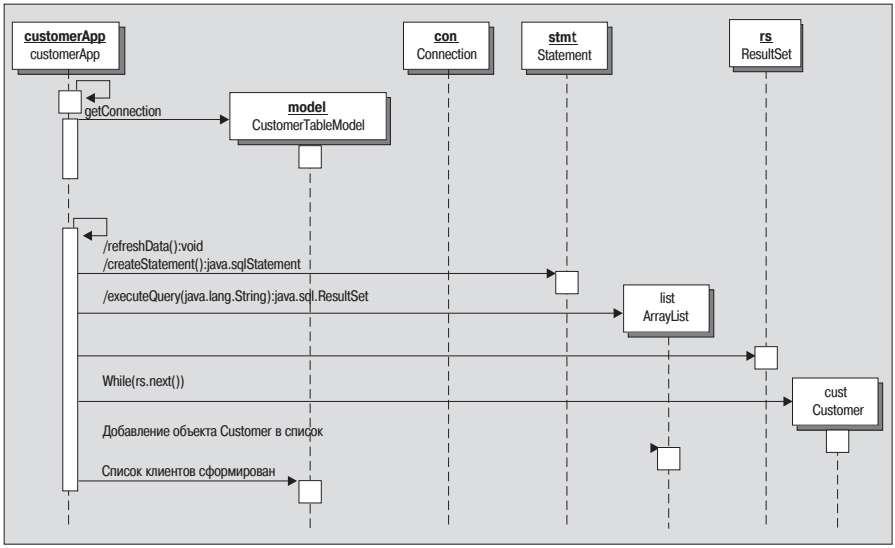


Рис. 17.5. Просмотр информации о клиентах

- Информация каждой записи результирующего множества применяется для создания объекта `Customer`, который добавляется в список
- Наконец, этот список нужен для заполнения модели, которая выдаст данные при помощи экземпляра `JTable`

Добавление нового клиента

Последовательность событий, вовлеченных в процесс добавления в базу данных нового клиента, изображена на рис. 17.6:

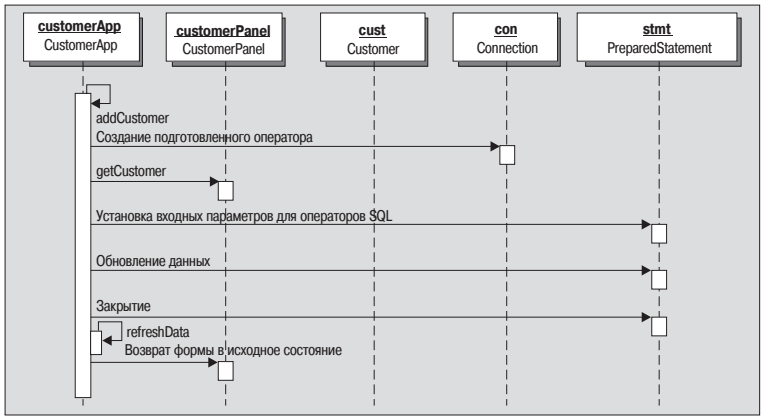


Рис. 17.6. Добавление нового клиента в базу данных

- Данные, введенные пользователем, извлекаются из экземпляра `CustomerPanel`, инкапсулированного в экземпляр класса `Customer`

- Соединение используется для создания подготовленного оператора
- Входные (IN) параметры для операторов SQL устанавливаются на основании данных, извлеченных из объекта Customer
- В заключение отправляется запрос на обновление, данные таблицы обновляются, и форма возвращается в исходное состояние

Удаление клиента

Диаграмма взаимодействия, приведенная ниже (рис. 17.7), показывает последовательность событий, происходящих при удалении клиента из базы данных:

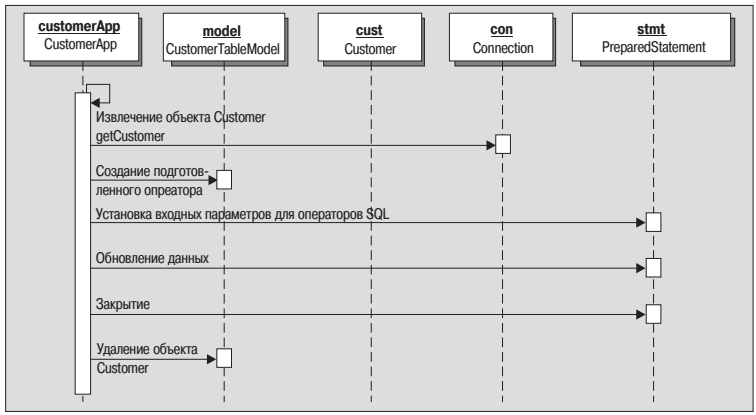


Рис. 17.7. Удаление клиента из базы данных

- Объект Customer, соответствующий выбранной строке таблицы, извлекается из модели
- Соединение используется для создания подготовленного оператора
- Входной параметр оператора SQL для идентификатора клиента устанавливается на основании данных, извлеченных из объекта Customer
- Отсылается запрос на обновление и соответствующий объект Customer удаляется из модели

Исходные файлы

В этом разделе подробно рассматриваются исходные файлы.

Класс Customer

Объявим переменные экземпляра для моделирования столбцов базы данных:

```
public class Customer {

    private int customerId = -1;
        private String title = "";
```

```
        private String fname = "";
        private String lname = "";
        private String addressLine = "";
        private String town = "";
    private String zipCode = "";
    private String phone = "";

    public Customer() {
    }
```

Конструктор для инициализации переменных экземпляра:

```
    public Customer(int customerId,String title,
        String fname,String lname,
        String addressLine,String town,
        String zipCode,String phone) {

        this.customerId = customerId;
        this.title = title;
        this.fname = fname;
        this.lname = lname;
            this.addressLine = addressLine;
            this.town = town;
            this.zipCode = zipCode;
            this.phone = phone;

    }
```

Получение и изменение идентификатора клиента:

```
    public int getCustomerId() {
        return customerId;
    }
    public void setCustomerId(int customerId) {
        this.customerId = customerId;
    }
```

Получение и изменение «титула»:

```
        public String getTitle() {
            return title;
        }

        public void setTitle(String title) {
            this.title = title;
        }
```

Получение и изменение имени:

```
    public String getFname() {
        return fname;
```

```
    }

    public void setFname(String fname) {
        this.fname = fname;
    }
```

Получение и изменение фамилии:

```
      public String getLname() {
            return lname;
        }

        public void setLname(String lname) {
            this.lname = lname;
        }
```

Получение и изменение адреса:

```
        public String getAddressLine() {
            return addressLine;
        }

        public void setAddressLine(String addressLine) {
            this.addressLine = addressLine;
        }
```

Получение и изменение города:

```
    public String getTown() {
            return town;
        }

        public void setTown(String town) {
            this.town = town;
        }
```

Получение и изменение почтового индекса:

```
        public String getZipCode() {
            return zipCode;
        }

        public void setZipCode(String zipCode) {
            this.zipCode = zipCode;
        }
```

Получение и изменение номера телефона:

```
        public String getPhone() {
            return phone;
        }
```

```
        public void setPhone(String phone) {
            this.phone = phone;
        }

}
```

Класс CustomerTableModel

Импортируем необходимые классы:

```
import javax.swing.table.AbstractTableModel;
import java.util.ArrayList;

public class CustomerTableModel extends AbstractTableModel {
```

Список для хранения текущих сущностей клиентов в базе данных:

```
    private ArrayList customerList = new ArrayList();
```

Задание списка клиентов. Если список изменяется, все необходимые объекты (listeners) уведомляются об этом. Уведомим таблицу о том, что она должна обновить саму себя:

```
    public void setCustomerList(ArrayList customerList) {
        this.customerList = customerList;
        fireTableDataChanged();
    }
```

Удаление определенного клиента. Если список изменяется, все необходимые объекты уведомляются об этом:

```
    public void removeCustomer(int index) {
        customerList.remove(index);
        fireTableDataChanged();
    }
```

Получение определенного клиента:

```
        public Customer getCustomer(int index) {

            if(index >= customerList.size()) {
                return null;
            }
            return (Customer)customerList.get(index);
        }
```

Метод обратного вызова для получения количества строк, которые должны быть выведены в таблице:

```
    public int getRowCount() {
        return customerList.size();
    }
```

Метод обратного вызова для получения количества столбцов, которые должны быть представлены в таблице:

```
    public int getColumnCount() {
        return 7;
    }
```

Метод обратного вызова для получения значения указанной ячейки, которое должно быть представлено в таблице:

```
    public Object getValueAt(int row,int col) {

        if(row >= customerList.size()) {
            throw new IllegalArgumentException("Invalid row");
        }

        Customer customer = (Customer)customerList.get(row);
        switch(col) {
            case 0:
                return customer.getTitle();
            case 1:
                return customer.getFname();
            case 2:
            return customer.getLname();
                case 3:
                      return customer.getAddressLine();
                case 4:
                      return customer.getTown();
                case 5:
                       return customer.getZipCode();
                case 6:
                       return customer.getPhone();
                default:
                       throw new IllegalArgumentException("Invalid column");
        }

    }

}
```

Класс CustomerPanel

Импортируем необходимые классы:

```
import javax.swing.JTextField;
import javax.swing.JPanel;
import javax.swing.JLabel;

import javax.swing.border.TitledBorder;
import javax.swing.border.EtchedBorder;
```

```
import java.awt.GridLayout;
import java.awt.FlowLayout;

public class CustomerPanel extends JPanel {
```

Создание полей для ввода сведений о новом клиенте:

```
    private JTextField title = new JTextField(15);
    private JTextField fname = new JTextField(15);
    private JTextField lname = new JTextField(15);
    private JTextField addressLine = new JTextField(15);
    private JTextField town = new JTextField(15);
    private JTextField zipCode = new JTextField(15);
    private JTextField phone = new JTextField(15);
```

Создание и возврат объекта `Customer` на основании значений, введенных в поля формы пользователем:

```
    public Customer getCustomer() {

        return new Customer(-1,title.getText(),
            fname.getText(),lname.getText(),
            addressLine.getText(),town.getText(),
            zipCode.getText(),phone.getText());

    }
```

Конструктор инициализирует пользовательский интерфейс для формы:

```
    public CustomerPanel() {
```

Создать сетку (grid) размером 7×1:

```
        setLayout(new GridLayout(7,1));
```

Добавляем поле для ввода «титула»:

```
        JPanel panel1 = new JPanel();
        panel1.setLayout(new FlowLayout(FlowLayout.RIGHT));
        panel1.add(new JLabel("Title:"));
        panel1.add(title);
        add(panel1);
```

Добавляем поле для ввода имени:

```
        JPanel panel2 = new JPanel();
        panel2.setLayout(new FlowLayout(FlowLayout.RIGHT));
        panel2.add(new JLabel("First Name:"));
        panel2.add(fname);
        add(panel2);
```

Добавляем поле для ввода фамилии:

```
        JPanel panel3 = new JPanel();
        panel3.setLayout(new FlowLayout(FlowLayout.RIGHT));
        panel3.add(new JLabel("Last Name:"));
        panel3.add(lname);
        add(panel3);
```

Добавляем поле для ввода адреса:

```
        JPanel panel4 = new JPanel();
        panel4.setLayout(new FlowLayout(FlowLayout.RIGHT));
        panel4.add(new JLabel("Address:"));
        panel4.add(addressLine);
        add(panel4);
```

Добавляем поле для ввода города:

```
        JPanel panel5 = new JPanel();
        panel5.setLayout(new FlowLayout(FlowLayout.RIGHT));
        panel5.add(new JLabel("Town:"));
        panel5.add(town);
        add(panel5);
```

Добавляем поле для ввода почтового индекса:

```
        JPanel panel6 = new JPanel();
        panel6.setLayout(new FlowLayout(FlowLayout.RIGHT));
        panel6.add(new JLabel("Zip Code:"));
        panel6.add(zipCode);
        add(panel6);
```

Добавляем поле для ввода номера телефона:

```
        JPanel panel7 = new JPanel();
        panel7.setLayout(new FlowLayout(FlowLayout.RIGHT));
        panel7.add(new JLabel("Phone:"));
        panel7.add(phone);
        add(panel7);

        setBorder(new TitledBorder(new EtchedBorder(),"Add Customer"));

    }
```

Этот метод очищает форму ввода клиентской информации:

```
    public void reset() {

        title.setText("");
        fname.setText("");
        lname.setText("");
```

```
            addressLine.setText("");
            town.setText("");
            zipCode.setText("");
            phone.setText("");

        }

    }
```

Класс CustomerApp

Импортируем необходимые классы:

```
    import javax.swing.JTable;
    import javax.swing.JFrame;
    import javax.swing.JScrollPane;
    import javax.swing.JPanel;
    import javax.swing.JButton;

    import java.awt.BorderLayout;

    import java.awt.event.ActionEvent;
    import java.awt.event.ActionListener;
    import java.awt.event.WindowEvent;
    import java.awt.event.WindowAdapter;

    import java.util.ArrayList;

    import java.sql.DriverManager;
    import java.sql.Connection;
    import java.sql.Statement;
    import java.sql.PreparedStatement;
    import java.sql.ResultSet;
    import java.sql.SQLException;

    public class CustomerApp extends JFrame implements ActionListener {
```

Экземпляр модели, питающей таблицу информации о клиентах:

```
        private CustomerTableModel model;
```

Таблица, содержащая информацию о клиентах:

```
        private JTable table;
```

Соединение с базой данных PostgreSQL:

```
        private Connection con;
```

Панель, содержащая форму ввода данных о клиенте:

```
    private CustomerPanel customerPanel;

    public CustomerApp(String[] arg) throws Exception {

        super("Customer Management System");
```

Установление соединения с базой данных:

```
        getConnection(arg[0],arg[1],arg[2]);
```

Создание экземпляра модели таблицы и заполнение ее данными:

```
        model = new CustomerTableModel();
        refreshData();
```

Создание таблицы при помощи модели и добавление таблицы в панель прокрутки. Затем панель добавляется в родительский фрейм:

```
        table = new JTable(model);
        table.setAutoCreateColumnsFromModel(true);
        JScrollPane pane = new JScrollPane(table);

        getContentPane().setLayout(new BorderLayout());

        getContentPane().add(pane,BorderLayout.CENTER);
```

Создаем две кнопки: одну – для добавления новых клиентов, другую – для удаления выделенного клиента. Текущий класс регистрируется как *получатель информации об изменениях (action listener)* двух кнопок:

```
        JPanel buttonPanel = new JPanel();
        JButton newButton = new JButton("Add Customer");
        newButton.addActionListener(this);
        buttonPanel.add(newButton);
        JButton deleteButton = new JButton("Remove Customer");
        deleteButton.addActionListener(this);
        buttonPanel.add(deleteButton);
        getContentPane().add(buttonPanel,BorderLayout.SOUTH);
```

Создаем экземпляр `CustomerPanel` и добавляем его во фрейм:

```
        customerPanel = new CustomerPanel();
        getContentPane().add(customerPanel,BorderLayout.WEST);
```

Отображаем фрейм:

```
        pack();
        show();
        setLocation(50,50);
```

```
        setSize(800,375);
        setResizable(false);

        validate();
```

Добавляем внутренний класс, получающий события закрытия окна и завершающий соединения с базой данных:

```
        addWindowListener(new WindowAdapter() {

            public void windowClosing(WindowEvent e) {
                try {
                    con.close();
                }catch(SQLException ex) {
                    ex.printStackTrace();
                }
                System.exit(0);
            }

        });

    }
```

Метод `main()` ожидает ввода JDBC URL, имени пользователя и пароля для доступа к базе данных в виде аргументов командной строки:

```
    public static void main(String args[]) throws Exception {

        if(args == null || args.length != 3) {
            System.out.print("Usage");
            System.out.println("java EmployeeApp
                <jdbcURL> <user> <passwd>");
            return;
        }
        CustomerApp app = new CustomerApp(args);

    }
```

Метод обратного вызова для прослушивания действий кнопок:

```
    public void actionPerformed(ActionEvent e) {

        JButton button = (JButton)e.getSource();

        try {
```

Если нажата кнопка добавления нового клиента, вызывается соответствующий метод:

```
            if("Add Customer".equals(button.getText())) {
                addCustomer();
```

Если нажата кнопка удаления клиента, вызывается соответствующий метод:

```
            }else if("Remove Customer".equals(button.getText())) {
                removeCustomer();
            }
        }catch(SQLException ex) {
            ex.printStackTrace();
        }

        validate();

    }
```

Этот метод устанавливает соединение с базой данных:

```
    private void getConnection(String url,String user,String passwd)
        throws Exception {

        Class.forName("org.postgresql.Driver");
        con = DriverManager.getConnection(url,user,passwd);

    }
```

Следующий метод обновляет данные модели таблицы:

```
    private void refreshData() throws SQLException {

        String sql = "select * from customer";
        Statement stmt = con.createStatement();
        ResultSet res = stmt.executeQuery(sql);

        ArrayList list = new ArrayList();
        while(res.next()) {
            Customer cust = new Customer(res.getInt(1),
                res.getString(2),res.getString(3),
                res.getString(4),res.getString(5),
                res.getString(6),res.getString(7),
                res.getString(8));
            list.add(cust);
        }

        model.setCustomerList(list);

        res.close();
        stmt.close();
    }
```

Этот метод вставляет запись в базу данных:

```
    private void addCustomer() throws SQLException {

        String sql = "insert into customer(" +
```

```
                        "title,fname,lname, " +
                        "addressline,town,zipcode,phone)" +
                          "values(" +
                          "?,?,?,?,?,?,?)";
```

Подготовка оператора:

```
            PreparedStatement stmt = con.prepareStatement(sql);

            Customer cust = customerPanel.getCustomer();
```

Установка входных параметров оператора SQL:

```
            stmt.setString(1,cust.getTitle());
            stmt.setString(2,cust.getFname());
            stmt.setString(3,cust.getLname());
            stmt.setString(4,cust.getAddressLine());
            stmt.setString(5,cust.getTown());
            stmt.setString(6,cust.getZipCode());
            stmt.setString(7,cust.getPhone());
```

Выполнение оператора SQL:

```
            stmt.executeUpdate();

            stmt.close();
```

Обновление таблицы:

```
            refreshData();
            customerPanel.reset();

        }
```

Этот метод удаляет запись из базы данных:

```
        private void removeCustomer() throws SQLException {

            String sql = "delete from customer where customer_id = ?";
            PreparedStatement stmt = con.prepareStatement(sql);
```

Получение указанного клиента:

```
            int selectedRow = table.getSelectedRow();

            Customer cust = model.getCustomer(selectedRow);
            if(cust == null) {
                return;
            }
```

Задание идентификатора клиента:

```
            stmt.setInt(1,cust.getCustomerId());
```

Выполнение оператора удаления SQL:

```
            stmt.executeUpdate();
            stmt.close();
```

Удаление клиента из модели:

```
            model.removeCustomer(selectedRow);

        }
    }
```

Компиляция и запуск приложения

Компилируем классы, используя следующую команду:

```
$ javac Customer*.java
```

Будут сгенерированы следующие файлы классов:

- `Customer.class`
- `CustomerTableModel.class`
- `CustomerPanel.class`
- `CustomerApp.class`

Запустим приложение такой командой:

```
$ java CustomerApp <JDBC URL> <User> <password>
```

Снимок экрана (рис. 17.8) показывает приложение в работе. Вызывая JVM, убедитесь, что файл `postgresql.jar` присутствует в пути классов системы.

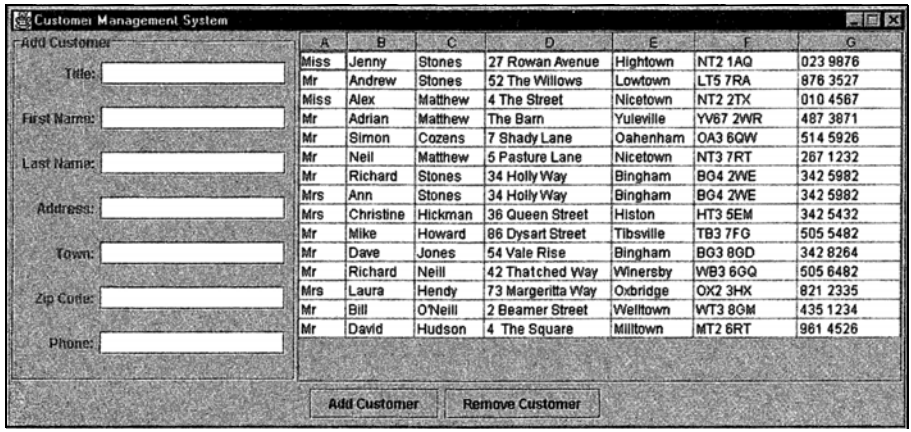


Рис. 17.8. Система управления клиентами

Резюме

В данной главе был рассмотрен доступ к базам данных PostgreSQL из программ, написанных на языке Java, с помощью JDBC. Обсуждались:

- Драйверы JDBC
- Сборка драйвера JDBC для PostgreSQL
- Соединения с базой данных. Операторы JDBC
- Подготовленные операторы для выполнения прекомпилированных команд SQL
- Результирующие множества JDBC
- Метаданные базы данных и результирующего множества

JDBC API продолжает развиваться, и когда производители перейдут к поддержке версии 3.0 JDBC API, станет доступен ряд новых интересных возможностей, в том числе:

- Поддержка точек сохранения
- Интеграция расширения API и базового API и реализация того и другого как в J2EE, так и в J2SE
- Метаданные параметров
- Возможность иметь одновременно несколько открытых результирующих множеств для оператора
- Усовершенствование вызываемых операторов

18

Дополнительная информация и ресурсы

Надеемся, что, прочитав книгу, вы по достоинству оценили PostgreSQL, эту передовую, обладающую массой возможностей реляционную базу данных. Начиная с версии 7.1 и далее PostgreSQL почти достигла соответствия основному стандарту SQL92, и кажется вероятным, что полное соответствие уже совсем близко.

Было показано, что доступ к PostgreSQL возможен из многих различных языков программирования, а также с удаленных машин, работающих по сети. Благодаря драйверу ODBC PostgreSQL может использоваться для сервера базы данных, даже если все клиентские машины будут работать под управлением Microsoft Windows.

В этой книге мы сконцентрировались на установке и работе PostgreSQL, в то время как некоторые ее выдающиеся возможности остались вне области рассмотрения, в частности создание таблиц, порожденных другими таблицами, с помощью наследования. Не было предпринято попытки тщательного изучения исходных текстов PostgreSQL, но все они, конечно же, доступны для просмотра и, если пожелаете, то можете изменить их для личного использования. Кто знает, может быть, однажды вы внесете свой вклад в разработку PostgreSQL.

Нереляционное хранилище

Реляционная модель существует уже около 30 лет, и за это время она показала себя как очень мощная и гибкая идея. Будучи абсолютно математически обоснованной, она выдержала проверку не только временем, но и суровой действительностью решения проблем реального мира.

За эти годы реляционной модели часто предъявлялись различные обвинения, самыми известными являются поступавшие со стороны объектных баз данных в 1990-х годах. Но несмотря на то что было несколько успешных чисто объектных баз данных, все же основная часть рынка осталась верна реляционной модели, во многом потому, что ведущие производители реляционных систем добавили в свои СУБД возможности, позволяющие упростить поддержку объектов.

Самый свежий вызов поступил (в свете установления XML в качестве формата обмена данными) от производителей, реализующих хранение и обработку XML в базе данных, основанной на XML. Такие базы данных характеризуются большой гибкостью, но похоже, что реляционная модель снова сможет приспособиться к ситуации, добавив XML-интерфейсы в традиционные базы данных, чтобы позволить им без труда использовать этот стандарт. Несмотря на то что формат XML очень гибок, передаваемые им данные почти всегда очень сильно структурированы и поддаются хранению в реляционной базе данных и извлечению из нее. Ожидается, что реляционная модель адаптируется и ответит на этот последний вызов.

OLTP, OLAP и другие термины базы данных

Мир баз данных изобилует профессиональным жаргоном: OLTP, OLAP, Data Warehouses, Data Marts и еще множество других терминов. Несмотря на то что они часто произносятся почти как магические заклинания, по существу это просто разные методики хранения данных, каждая из которых оптимизирована для определенных целей.

В этой книге реляционная база данных использовалась очень интерактивно. Она воспринималась как источник реальных данных, обновляющийся по мере поступления заказов и т. д., к которому одновременно имеют доступ несколько пользователей, имеющие возможность вносить свои изменения. Все это характеристики **обработки транзакций в реальном времени** (On Line Transaction Processing, OLTP).

Существует другой класс данных, используемых не для интерактивной обработки, а для анализа произошедших событий. В этой связи часто употребляются термины: система поддержки принятия решений (Decision Support Systems, DSS), информационные хранилища (Data Warehouses), управленческая информационная система (Executive Information Systems, EIS) и административная информационная система (Management Information System, MIS). Мы будем оперировать технически более строгим понятием **аналитической обработки в реальном времени** (On Line Analytical Processing, OLAP). Требования к таким системам сильно отличаются от предъявляемых к более широко распространенным базам данных типа OLTP.

Некоторые существенные отличия между базами данных разных типов представлены в табл. 18.1:

Таблица 18.1. Базы данных OLAP и OLTP

OLTP	OLAP
Реальные данные, обычно продолжающие обновляться	Исторические данные, обновления в фиксированное время, например еженощно или реже
Используются для обработки во время работы	Используется для анализа за долгий промежуток времени
Размер обычно ограничен несколькими десятками гигабайтов	Размер может составлять несколько терабайтов
Обычно содержит ограниченный объем исторических данных	Может хранить исторические данные за многие годы
Оптимизирована в реляционной модели для эффективного обновления данных	Оптимизирована для извлечения информации, обычно в нереляционной модели, такой как радиальная

Примеры, приведенные в данной книге, основывались на очень простой базе данных `bpsimple`, в которой хранились клиенты, размещенные ими заказы и продаваемые товары. Схема этой базы данных принадлежит к типу OLTP. Она была оптимизирована как реляционная база данных для обеспечения динамической обработки заказов и эффективного хранения данных.

Если бы мы успешно занимались крупной коммерческой деятельностью, то через некоторое время объем данных стал бы весьма велик, и производительность начала бы падать. Если быть реалистами, то надо признать, что лучший способ быстрого увеличения производительности состоит в простом уничтожении старых данных. Вполне можно решить, что клиенты, не делавшие заказов в течение двух лет, нас уже не интересуют, как и товары, отгруженные год назад.

Однако можно попытаться провести анализ привычек клиентов и их заказов, чтобы усовершенствовать планирование сбыта. Вероятно, можно провести целевую маркетинговую или рекламную кампанию, чтобы продать больше товаров.

Расскажем классическую историю о супермаркете, в котором обнаружили, что между продажами детских подгузников и пива существует связь, и что, поместив их в зале друг напротив друга, можно увеличить продажи. Объяснялось это тем, что некоторые мужья, которым было дано задание купить подгузники по дороге с работы, видя стоящее рядом пиво, соблазнялись возможностью заодно приобрести пару бутылочек.

Для такого рода исследований требуется огромное количество данных, вероятно, собираемых годами, а также необходима возможность эф-

фективного обращения к ним. Не имеет значения, что вчерашние цифры или продажи прошлых выходных еще не внесены в базу, ведь нас интересуют долгосрочные тенденции. Этой работой и занимаются базы данных OLAP. Хотя они могут надстраиваться (так часто и бывает) над обычными реляционными базами данных, но хранимые данные организованы так, чтобы их извлечение было максимально эффективным.

В основе базы данных OLAP часто лежит радиальная схема, с центральной таблицей **фактов** (fact table) и несколькими связанными с ней таблицами **измерений** (dimension table). Создавая базу данных OLAP для нашего хранилища, вероятно, следовало бы сделать продажи центральной таблицей фактов, а все остальные данные: о тех, кто купил товары, о дате продаж, а также сведения о товарах можно было бы отнести в разряд таблиц измерений (рис. 18.1):

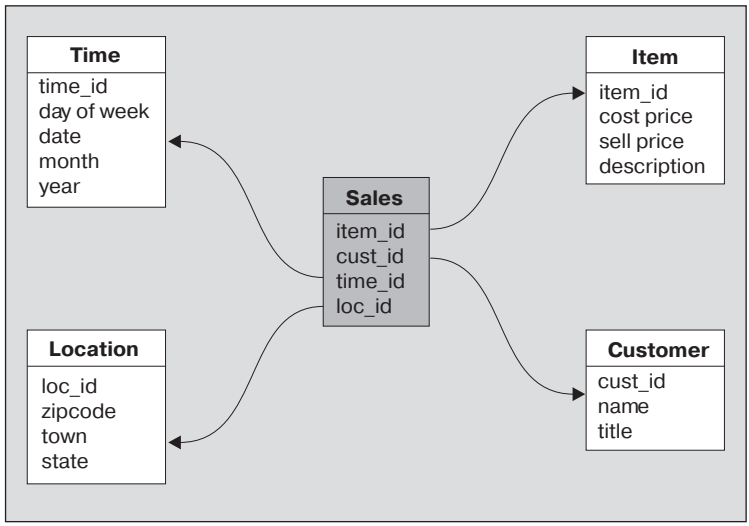


Рис. 18.1. Радиальная схема для магазина

Для того чтобы данные попали в такую базу данных OLAP, вероятно, пришлось бы экспортировать их из обычной базы данных, а затем применить специальные процедуры загрузки данных для встраивания данных в новый формат.

Если бы у нас был очень крупный бизнес, то могла бы возникнуть потребность разбить базу данных OLAP на более мелкие и более специализированные сегменты данных. Их часто называют информационными гиперкубами (data cubes) или витринами данных (data marts). Это меньшие совокупности данных об отдельных фактах, организованные как база данных OLAP.

Поэтому общий поток данных крупной организации с большими потребностями обработки в реальном времени и необходимостью тонкого анализа данных может быть изображен, как на рис. 18.2:

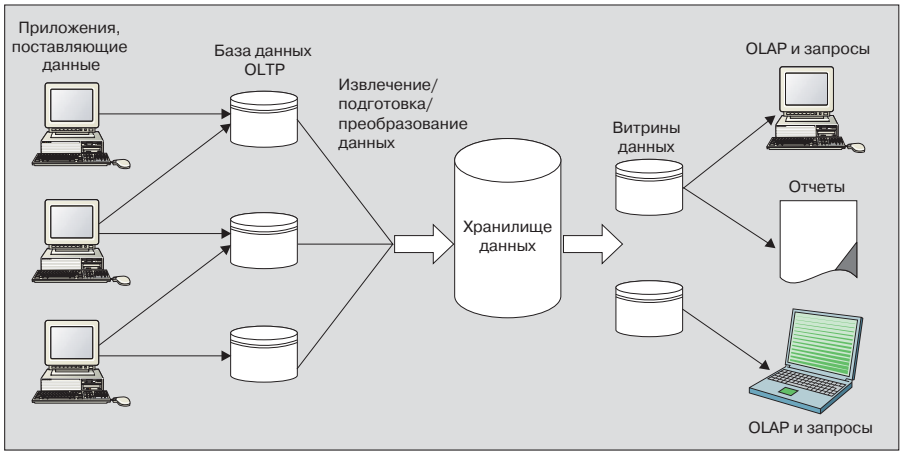


Рис. 18.2. Организация хранения и обработки данных в большой компании

Хотя схема таких баз данных OLAP очень отличается от более стандартных реляционных схем, тем не менее они обычно реализуются при помощи реляционных баз данных. Главные производители, такие как Oracle и Microsoft, ввели в своих базах данных дополнительные возможности, чтобы упростить работу с ними как с основой для баз данных OLAP.

Ресурсы

Благодаря тому что реляционная модель существует уже давно, в настоящее время доступен большой объем справочных материалов о ней.

Веб-ресурсы

Веб-ресурсы, конечно же, все время меняются, но все же упомянем несколько полезных отправных точек:

PostgreSQL

- *http://www.postgresql.org*
- *http://www.unixodbc.org*
- *http://www.greatbridge.org*
- *http://pgdemo.acucore.com*
- *http://www.pgsql.com*
- *http://www.redhat.com* и поиск «Red Hat Database»

PHP

- *http://www.php.net*
- *http://www.phpbuilder.com*
- *http://www.phpwizard.net*

Perl

- *http://www.perl.com*
- *http://www.perl.org*
- *http://www.cpan.org*
- *http://activestate.com*

Java и JDBC

- *http://www.java.sun.com*

Общий инструментарий

«DeZign for databases» – это среда разработки баз данных с использованием диаграммы отношений. Она поддерживает графическое представление схемы объектов и отношений и автоматически генерирует схемы на SQL для большинства ведущих баз данных. Поддерживает диаграммы «объекты-отношения» (Entity Relationship Diagrams, ERD), работает в Windows. Доступна пробная версия:

- *http://www.datanamic.com/dezign/index.html*

«Toolkit for Conceptual Modeling» (средства концептуального моделирования) представляет собой набор программных средств для представления концептуальных моделей систем программного обеспечения в форме диаграмм, таблиц и деревьев. Включает в себя ERD, но (пока) не генерирование кода. Работает в UNIX и Linux:

- *http://wwwhome.cs.utwente.nl/~tcm/*

«Database Design Studio» (DDS) – это инструмент проектирования баз данных на основе диаграмм «объекты-отношения». Работает в Windows с помощью ODBC:

- *http://www.chillisource.com/dds/*

Книги

SQL

- «PostgreSQL – Introduction and Concepts» (PostgreSQL – введение и концепции), Bruce Momjian (Брюс Момжиан). Addison-Wesley (ISBN 0-201-70331-9). Книга одного из основных разработчиков PostgreSQL. Также доступна в формате HTML в Интернете.
- «Database Design for Mere Mortals – A Hands-On guide to Relational Database Design» (Проектирование баз данных для простых смерт-

ных – практическое руководство по проектированию реляционных баз данных), Michael J. Hernandez (Майкл Д. Хернандес). Addison-Wesley (ISBN 0-201-69471-9). Книга посвящена сбору информации и основам проектирования баз данных, написана в легкой для чтения манере.

- «The Practical SQL Handbook – Using Structured Query Language» (Практическое справочное руководство по SQL – Использование структурированного языка запросов), Judith S. Bowman (Джудит С. Боуман), Sandra L. Emerson (Сандра Л. Эмерсон), Marcy Darnovsky (Марси Дарновски). Addison-Wesley (ISBN 0-201-44787-8). Четкие объяснения некоторых практических аспектов использования SQL.
- «Mastering SQL» (Освоение SQL), Martin Gruber (Мартин Грубер). Sybex (ISBN 0-7821-2538-7). Хорошо написанный справочник по стандартному SQL.
- «SQL for Smarties» (SQL для профессионалов), Joe Celko (Джо Селко). Morgan Kaufmann Publishers (ISBN 1-55860-576-2). Великолепная книга, представляющая массу возможностей SQL для самых опытных пользователей.
- «Instant SQL Programming» (Повседневное программирование на SQL), Joe Celko (Джо Селко). Wrox Press (ISBN 1-874416-50-8). Простое руководство по синтаксису и применению SQL.

PHP

- «Professional PHP Programming» (Профессиональное PHP-программирование), Jesus Castagnetto (Джезус Кастаньетто), Chris Scollo (Крис Сколло), Sascha Schumann (Саша Шуман), Harish Rawat (Хэриш Рават), Deepak Veliath (Дипак Велиаф). Wrox Press (ISBN 1-861002-96-3). Это книга о программировании в многозвенной архитектуре на PHP, включающая введение в PHP и обращение к базам данных.
- «Beginning PHP4» (Введение в PHP4), Wankyu Choi (Ванкиу Чой), Allan Kent (Алан Кент), Ganesh Prasad (Гэниш Прасад) и Chris Ullman (Крис Уллман), при участии Jon Blank (Джона Бланка) и Sean Cazzell (Шона Кэзела). Wrox Press (ISBN 1861003730). Учебное пособие по языку PHP, содержащее введение в реляционные базы данных.

Perl

- «Beginning Perl» (Введение в Perl), Simon Cozens (Саймон Казенс). Wrox Press (ISBN 1861003439). Руководство по Perl для Windows и UNIX, включающее работу с базами данных.
- «Perl Cookbook. Third Edition» (Perl: книга рецептов. Третье издание), Tom Christiansen (Том Кристиансен) и Nathan Torkington (Натан Торкингтон). O'Reilly and Associates (ISBN 0-596-00027-8). Это издание охватывает Perl 5.6 и все остальные основные элементы.

- «Professional Perl Programming» (Профессиональное программирование на Perl), Peter Wainwright (Питер Уэйнрайт), а также Aldo Caplini (Алдо Каплини), Simon Cozens (Симон Козенс), JJ Merelo-Guervos (ЖЖ Мерело-Гуэрвос), Aalhad Saraf (Аалхад Сараф) и Chris Nandor (Крис Нандор). Wrox Press (ISBN 1-861004-49-4). Книга содержит углубленное описание Perl 5.6, объектно-ориентированного программирования и многого другого.

Java

- «Beginning Java 2» (Введение в Java 2), Ivor Horton (Ивор Хортон). Wrox Press (ISBN1-861003-66-8). Книга для каждого, кто хочет программировать на Java. Она с нуля учит языку Java, а также объектно-ориентированному программированию, соединению с базами данных с помощью JDBC и т. д.
- «Professional Java Data» (Сведения о Java для профессионалов), Danny Ayers (Дэнни Айерс), John Bell (Джон Белл), Carl Calvert-Bettis (Карл Калверт-Беттис), Thomas Bishop (Томас Бишоп), Bjarki Holm (Бьярки Холм), Glenn E. Mitchell (Гленн Э. Митчелл), Kelly Lin Poon (Келли Лин Пун), Sean Rhody (Шон Роди), а также Mike Bogovich (Майк Боговић), Matthew Ferris (Мэтью Феррис), Rick Grehan (Рик Грэхан), Tony Loton (Тони Лотон), Nitin Nanda (Нитин Нанда) и Mark Wilcox (Марк Уилкокс). Wrox Press (ISBN 1-861004-10-9). Информация о доступе к данным реляционных и объектно-ориентированных баз данных при помощи Java.

Резюме

Со времени начала исследований PostgreSQL сделала значительные успехи и превратилась в устойчивый продукт, обладающий большими возможностями, подходящий для применения во многих производственных системах. Она достаточно строго следует стандарту SQL и продолжает совершенствоваться с каждой редакцией, становясь все более масштабируемой.

Лицензия Open Source означает, что можно использовать PostgreSQL, не платя ни за клиентскую, ни за серверную лицензии. Благодаря бесплатности лицензии, а также знаниям, полученным из этой книги, вы теперь должны суметь развернуть настоящий сервер базы данных в тех ситуациях, когда лицензионные выплаты заставили бы вас принять менее изящное решение.

У PostgreSQL уже есть выдающееся прошлое, и т. к. она быстро развивается, превращаясь в очень стабильную и мощную базу данных, то мы уверены, что она составит достойную конкуренцию производителям коммерческих баз данных с их огромными бюджетами на разработку и специальными группами разработчиков. Очень этого хотелось бы.

A

Ограничения базы данных PostgreSQL

Используя базу данных для хранения информации, создавая таблицы и добавляя в них строки, мы склонны игнорировать тот факт, что ни на одной платформе невозможно существование бесконечного хранилища.

Все базы данных так или иначе ограничены, и PostgreSQL не исключение. Объем данных, которые могут быть сохранены в одном столбце, максимально разрешенное для таблицы количество строк, общий размер таблицы – все это имеет свои пределы, хотя и немалые.

В последних версиях PostgreSQL многие ограничения ослаблены, а во многих случаях действительно удалены. Упомянем здесь некоторые из еще действующих в PostgreSQL 7.1 ограничений.

За обновлениями, касающимися предельных значений для более поздних версий, можно следить на сайте *http://www.postgresql.org*.

> ***Информация извлечена из PostgreSQL FAQ и из списка рассылки ее разработчиков.***

Если о размере сказано «нет ограничений», это означает, что сам PostgreSQL не накладывает никаких рамок. Максимальный размер будет определяться другими факторами, такими как объем свободного дискового пространства или виртуальной памяти.

Приближение к предельным значениям сопровождается уменьшением производительности базы данных. Если, например, обрабатывать очень большие поля, поглощающие значительную часть доступной (виртуальной) памяти, вполне возможно, что производительность ста-

нет неприемлемой. В конце концов, PostgreSQL просто физически не сможет выполнить обновление.

Другие, не обсуждаемые здесь ограничения, накладываются операционной системой или же способом передачи по сети. Например, существуют типичные предельные размеры запроса, который можно осуществить через ODBC, зависящие от драйвера. Могут применяться и ограничения памяти, которые не допустят, чтобы очень большие столбцы, строки или результирующие множества создавались, передавались по сети (которая тоже будет перегружена) или получались клиентом.

Размер базы данных: нет ограничений

PostgreSQL не накладывает ограничения на суммарный размер базы данных. Известны базы данных, превышающие 60 Гбайт. База данных размером 60 Гбайт более чем достаточна для самых требовательных приложений.

Из-за особенностей организации хранилища данных PostgreSQL можно заметить некоторое ухудшение производительности для баз данных, содержащих много таблиц. PostgreSQL использует много файлов для хранения табличных данных, и если операционная система не очень хорошо справляется с большим количеством файлов в одном каталоге, то производительность может упасть.

Размер таблицы: 16 Тбайт – 64 Тбайт

Обычно PostgreSQL хранит свои данные порциями по 8 Кбайт. Количество таких блоков ограничено 32-разрядным целым числом со знаком (около 2 миллиардов), что дает максимальный размер таблицы 16 Тбайт. Размер базового блока может быть увеличен при сборке PostgreSQL до максимума в 32 Кбайт, тогда теоретический предел размера таблицы возрастет до 64 Тбайт.

Некоторые операционные системы накладывают ограничения на размер файла, не позволяющие создавать файлы большего размера, поэтому PostgreSQL хранит табличные данные в нескольких файлах, размером по 1 Гбайт каждый. В случае больших таблиц это приводит к появлению значительного количества файлов и потенциальному ухудшению производительности операционной системы, о чем уже говорилось в этой книге.

Строки таблицы: нет ограничений

PostgreSQL не накладывает ограничений на количество строк в какой бы то ни было таблице.

Но агрегатная функция COUNT возвращает 32-разрядное целое число. Поэтому для таблиц, содержащих более двух миллиардов строк, результат COUNT будет бессмысленным.

Индексы таблиц: нет ограничений

PostgreSQL не накладывает ограничений на количество индексов, создаваемых для таблицы. Конечно, если создавать все больше и больше индексов для таблицы со все большим и большим количеством столбцов, то производительность будет ухудшаться.

Размер столбца: 1 Гбайт

PostgreSQL определяет предел в один гигабайт для размера любого одного поля таблицы. На практике этот предел определяется объемом памяти, доступной серверу для манипулирования данными и передачи их клиенту.

Столбцов в таблице: 250 и более

Максимальное количество столбцов, которые могут быть помещены в таблицу PostgreSQL, зависит от сконфигурированного размера блока и от типа столбца. Для размера блока по умолчанию (8 Кбайт) могут быть сохранены как минимум 250 столбцов. Их количество может возрасти до 1600, если все столбцы представляют собой очень простые поля, например, являются целыми числами. Увеличение размера блока соответственно увеличивает эти предельные значения.

Размер строки: нет ограничений

Не существует явно выраженного максимального размера строки. Но размер столбцов и их количество ограничены, как описано выше.

B

Типы данных PostgreSQL

PostgreSQL поддерживает достаточно большой набор типов, которые описаны в руководстве пользователя. О них также можно получить информацию, введя команду `\dT` в `psql`. Представим здесь лишь самые необходимые типы данных, а наиболее специализированные и внутренние типы PostgreSQL пропустим.

В таблицах в первом столбце приведено стандартное название SQL, обычно используемое и в PostgreSQL, а во втором – альтернативные названия, принятые в PostgreSQL.

Некоторые типы являются специальными типами PostgreSQL. В таких случаях название SQL не указывается. Если это возможно, то рекомендуем придерживаться стандартных типов и названий SQL.

Логические типы

Название в SQL	Альтернативное название PostgreSQL	Замечания
`boolean`	`bool`	Хранит логическое значение. Принимает в качестве «истины» следующие значения: `TRUE`, `'t'`, `'true'`, `'y'`, `'yes'`, `'1'`. Использует 1 байт памяти, может хранить `NULL`, в отличие от некоторых коммерческих баз данных. Логический тип был добавлен в стандарт только в SQL99, хотя повсеместно использовался задолго до этого.

Типы точных чисел

Название в SQL	Альтернативное название PostgreSQL	Замечания
smallint	int2	Двухбайтное целое со знаком, может хранить значения от -32768 до +32767.
integer, int	int4	Четырехбайтное целое со знаком, может хранить числа от -2147483648 до +2147483647.
	int8	Восьмибайтное целое со знаком, может хранить приблизительно 18 десятичных разрядов.
bit		Хранит один бит (0 или 1).
bit varying	varbit	Хранит битовую последовательность. Чтобы вставить в таблицу, используйте такой синтаксис: INSERT INTO … VALUES(011101::varbit);.

Типы приблизительных чисел

Название в SQL	Альтернативное название PostgreSQL	Замечания
numeric (precision, scale)		Хранит число с указанной точностью (precision). Руководство пользователя утверждает, что ограничения точности не существует.
decimal (precision, scale)		По умолчанию точность равна 9, а масштаб (scale) – 0. В руководстве пользователя сказано, что диапазон приблизительно равен 8000 разрядов. В стандартном SQL разница между десятичным (decimal) и числовым (numeric) типами заключается в том, что для числового типа точность должна быть равна той, что указана, а для десятичного реализация может определить дополнительную точность. Рекомендуем придерживаться числового типа и не применять десятичный.
float (precision)	float4, float8	Число с плавающей точкой с как минимум указанной точностью. Если требуемая точность не превышает 7 разрядов, то следует выбрать float4, иначе – float8 с максимальной точностью в 15 разрядов. Используйте float(15), чтобы получить эквивалент стандартного типа SQL double precision.
real	float4	Советуем предпочесть float(precision).
double precision	float8	Аналог float(15).
	money	Аналог decimal(9,2). Его применение не одобряется, т. к. он исключен из числа рекомендуемых, и его поддержка может быть удалена из следующих версий PostgreSQL.

Типы даты/времени

Название в SQL	Альтернативное название PostgreSQL	Замечания
`timestamp`	`datetime`	Хранит значения времени от 4713 г. до н. э. до 1 465 001 г. н. э., с разрешением в 1 микросекунду.
`timestamp with timezone`		Хранит значения времени от 1903 г. н. э. до 2037 г. н. э., с разрешением в 1 микросекунду.
`interval`	`interval, timespan`	Может хранить временной промежуток приблизительно в +/– 178 000 000 лет, с разрешением в 1 микросекунду.
`date`		Хранит даты от 4713 г. до н. э. до 32 767 г. н. э. с разрешением в 1 день.
`time`		Хранит время суток от 0 до 23:59:59.99, с разрешением в 1 микросекунду.
`time with timezone`		Аналогично `time`, но хранится еще и часовой пояс.

Символьные типы

Название в SQL	Альтернативное название PostgreSQL	Замечания
`char`		Хранит один символ.
`char(n)`		Хранит `n` символов, если сохраняется меньше символов, они будут дополнены пробелами. Рекомендуется только для коротких строк фиксированной длины.
`char varying(n)`	`varchar(n)`	Хранит переменное количество символов, до максимума в `n` символов, которые не дополняются пробелами. Это «стандартный» выбор для символьных строк.
	`text`	Специальный вариант PostgreSQL для `varchar`, не требующий указания верхнего предела количества символов.

Геометрические типы

Название в SQL	Альтернативное название PostgreSQL	Замечания
	point	Значение x,y
	line	Линия (pt1, pt2)
	lseg	Отрезок линии (pt1, pt2)
	box	Прямоугольник, задаваемый парой точек
	path	Последовательность точек, может быть открытой или закрытой
	polygon	Последовательность точек, представляет собою замкнутый путь, но он по-другому обрабатывается внутри PostgreSQL
	circle	Точка и радиус, задающие окружность

Разные типы

Название в SQL	Альтернативное название PostgreSQL	Замечания
serial		В стандартном SQL serial это числовой столбец, значение которого возрастает при добавлении каждой строки. В PostgreSQL serial не реализован как отдельный тип, хотя стандартный синтаксис SQL принимается. Внутри же себя PostgreSQL использует integer для хранения значения и последовательность для управления автоматическим увеличением значения. При создании таблицы с типом serial создается неявная последовательность для контроля за данными столбца такого типа. Эта неявная последовательность не удаляется автоматически при удалении таблицы.
	oid	Идентификатор объекта. Внутри PostgreSQL скрытый oid добавляется к каждой строке и хранит четырехбайтное целое, что дает максимальное значение приблизительно в 4 миллиарда. Этот тип также применяется как ссылка при хранении больших бинарных объектов.

Название в SQL	Альтернативное название PostgreSQL	Замечания
	`cidr`	Хранит сетевые адреса в виде x.x.x.x/y, где y – этом маска сети. CIDR – это Classless Inter-Domain Routing (бесклассовая междоменная маршрутизация). В обычном IP есть три класса: A, B и C, сетевая часть которых состоит из соответственно 8, 16 и 24 битов, что допускает наличие 16.7 миллионов, 65 тысяч и 254 хостов в одной сети. CIDR разрешает маски сети любого размера, благодаря чему можно лучше распределить IP-адреса и маршруты между ними иерархически.
	`inet`	Аналогичен `cidr`, но адрес хоста может быть равен 0.
	`macaddr`	MAC-адрес в виде XX:XX:XX:XX:XX:XX.

C

Синтаксис SQL в PostgreSQL

Команды SQL в PostgreSQL

ABORT
ALTER GROUP
ALTER TABLE
ALTER USER
BEGIN
CHECKPOINT
CLOSE
CLUSTER
COMMENT
COMMIT
COPY
CREATE AGGREGATE
CREATE CONSTRAINT TRIGGER
CREATE DATABASE
CREATE FUNCTION
CREATE GROUP
CREATE INDEX
CREATE LANGUAGE
CREATE OPERATOR
CREATE RULE
CREATE SEQUENCE
CREATE TABLE
CREATE TABLE AS
CREATE TRIGGER
CREATE TYPE
CREATE USER
CREATE VIEW
DECLARE
DELETE
DROP AGGREGATE
DROP DATABASE
DROP FUNCTION
DROP GROUP
DROP INDEX
DROP LANGUAGE
DROP OPERATOR
DROP RULE
DROP SEQUENCE
DROP TABLE
DROP TRIGGER
DROP TYPE
DROP USER
DROP VIEW
END
EXPLAIN
FETCH
GRANT
INSERT
LISTEN
LOAD
LOCK
MOVE
NOTIFY
REINDEX
RESET
REVOKE
ROLLBACK
SELECT
SELECT INTO
SET
SET CONSTRAINTS
SET TRANSACTION
SHOW
TRUNCATE
UNLISTEN
UPDATE
VACUUM

Синтаксис SQL в PostgreSQL

ABORT

```
ABORT [ WORK | TRANSACTION ]
```

Описание: прекращает текущую транзакцию

ALTER GROUP

```
ALTER GROUP name ADD USER username [, ... ]
ALTER GROUP name DROP USER username [, ... ]
```

Описание: добавляет пользователей в группу, удаляет пользователей из группы

ALTER TABLE

```
ALTER TABLE [ ONLY ] table [ * ]
    ADD [ COLUMN ] column type
ALTER TABLE [ ONLY ] table [ * ]
    ALTER [ COLUMN ] column { SET DEFAULT value | DROP DEFAULT }
ALTER TABLE table [ * ]
    RENAME [ COLUMN ] column TO newcolumn
ALTER TABLE table
    RENAME TO newtable
ALTER TABLE table
    ADD table constraint definition
ALTER TABLE table
    OWNER TO new owner
```

Описание: изменяет свойства таблицы

ALTER USER

```
ALTER USER username
    [ WITH PASSWORD 'password' ]
    [ CREATEDB | NOCREATEDB ] [ CREATEUSER | NOCREATEUSER ]
    [ VALID UNTIL 'abstime' ]
```

Описание: изменяет свойства учетной записи пользователя

BEGIN

```
BEGIN [ WORK | TRANSACTION ]
```

Описание: начинает транзакцию в режиме явных транзакций

CHECKPOINT

```
CHECKPOINT
```

Описание: заставляет транзакцию регистрировать контрольные точки

CLOSE

```
CLOSE cursor
```

Описание: закрывает курсор

CLUSTER

```
CLUSTER indexname ON tablename
```

Описание: дает серверу подсказку о кластеризации хранилища

COMMENT

```
COMMENT ON
[
  [ DATABASE | INDEX | RULE | SEQUENCE | TABLE | TYPE | VIEW ]
  object_name |
  COLUMN table_name.column_name|
  AGGREGATE agg_name agg_type|
  FUNCTION func_name (arg1, arg2, ...)|
  OPERATOR op (leftoperand_type rightoperand_type) |
  TRIGGER trigger_name ON table_name
] IS 'text'
```

Описание: добавляет комментарий для объекта

COMMIT

```
COMMIT [ WORK | TRANSACTION ]
```

Описание: фиксирует текущую транзакцию

COPY

```
COPY [ BINARY ] table [ WITH OIDS ]
    FROM { 'filename' | stdin }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]
COPY [ BINARY ] table [ WITH OIDS ]
    TO { 'filename' | stdout }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]
```

Описание: копирует данные из файлов в таблицы и обратно

CREATE AGGREGATE

```
CREATE AGGREGATE name ( BASETYPE = input_data_type,
    SFUNC = sfunc, STYPE = state_type
    [ , FINALFUNC = ffunc ]
    [ , INITCOND = initial_condition ] )
```

Описание: определяет новую агрегатную функцию

CREATE CONSTRAINT TRIGGER

```
CREATE CONSTRAINT TRIGGER name
    AFTER events ON
    relation constraint attributes
    FOR EACH ROW EXECUTE PROCEDURE func '(' args ')'
```

Описание: создает триггер, который может использоваться для реализации ограничений

CREATE DATABASE

```
CREATE DATABASE name
    [ WITH [ LOCATION = 'dbpath' ]
           [ TEMPLATE = template ]
           [ ENCODING = encoding ] ]
```

Описание: создает новую базу данных

CREATE FUNCTION

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    AS definition
    LANGUAGE 'langname'
    [ WITH ( attribute [, ...] ) ]
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    AS obj_file , link_symbol
    LANGUAGE 'langname'
    [ WITH ( attribute [, ...] ) ]
```

Описание: определяет новую функцию

CREATE GROUP

```
CREATE GROUP name
    [ WITH
     [ SYSID gid ]
     [ USER  username [, ...] ] ]
```

Описание: создает новую группу

CREATE INDEX

```
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_name ] ( column [ ops_name ] [, ...] )
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_name ] ( func_name( column [, ... ]) [ ops_name ] )
```

Описание: создает дополнительный индекс

CREATE LANGUAGE

```
CREATE [ TRUSTED ] [ PROCEDURAL ] LANGUAGE 'langname'
    HANDLER call_handler
    LANCOMPILER 'comment'
```

Описание: определяет дополнительный язык для функций

CREATE OPERATOR

```
CREATE OPERATOR name ( PROCEDURE = func_name
     [, LEFTARG = type1 ] [, RIGHTARG = type2 ]
     [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
     [, RESTRICT = res_proc ] [, JOIN = join_proc ]
     [, HASHES ] [, SORT1 = left_sort_op ] [, SORT2 =
                right_sort_op ] )
```

Описание: определяет новый пользовательский оператор

CREATE RULE

```
CREATE RULE name AS ON event
    TO object [ WHERE condition ]
    DO [ INSTEAD ] action
```

где `action` (действие) может быть таким:

```
NOTHING
|
query
|
( query ; query ... )
|
[ query ; query ... ]
```

Описание: определяет новое правило

CREATE SEQUENCE

```
CREATE SEQUENCE seqname [ INCREMENT increment ]
    [ MINVALUE minvalue ] [ MAXVALUE maxvalue ]
    [ START start ] [ CACHE cache ] [ CYCLE ]
```

Описание: создает новый генератор последовательности

CREATE TABLE

```
CREATE [ TEMPORARY | TEMP ] TABLE table_name (
    { column_name type [ column_constraint [ ... ] ]
      | table_constraint }  [, ... ]
    ) [ INHERITS ( inherited_table [, ... ] ) ]
```

где column_constraint (ограничение для столбца) может быть:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | UNIQUE | PRIMARY KEY | DEFAULT value | CHECK (condition) |
  REFERENCES table [ ( column ) ] [ MATCH FULL | MATCH PARTIAL ]
   [ ON DELETE action ] [ ON UPDATE action ]
   [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
}
```

table_constraint (ограничение для таблицы) может быть:

```
[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ... ] ) |
  PRIMARY KEY ( column_name [, ... ] ) |
  CHECK ( condition ) |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES table [ ( column [, ... ] ) ]
   [ MATCH FULL | MATCH PARTIAL ] [ ON DELETE action ] [ ON UPDATE action ]
   [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE
]
}
```

Описание: создает новую таблицу

CREATE TABLE AS

```
CREATE TABLE table [ (column [, ...] ) ]
     AS select_clause
```

Описание: создает новую таблицу

CREATE TRIGGER

```
CREATE TRIGGER name { BEFORE | AFTER } { event [OR ...] }
    ON table FOR EACH { ROW | STATEMENT }
    EXECUTE PROCEDURE func ( arguments )
```

Описание: создает новый триггер

CREATE TYPE

```
CREATE TYPE typename ( INPUT = input_function, OUTPUT = output_function
      , INTERNALLENGTH = { internallength | VARIABLE }
    [ , EXTERNALLENGTH = { externallength | VARIABLE } ]
    [ , DEFAULT = "default" ]
    [ , ELEMENT = element ] [ , DELIMITER = delimiter ]
    [ , SEND = send_function ] [ , RECEIVE = receive_function ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
)
```

Описание: создает новый базовый тип данных

CREATE USER

```
CREATE USER username
    [ WITH
     [ SYSID uid ]
     [ PASSWORD 'password' ] ]
    [ CREATEDB   | NOCREATEDB ] [ CREATEUSER | NOCREATEUSER ]
    [ IN GROUP     groupname [, ...] ]
    [ VALID UNTIL  'abstime' ]
```

Описание: создает нового пользователя базы данных

CREATE VIEW

```
CREATE VIEW view AS SELECT query
```

Описание: создает новое представление

DECLARE

```
DECLARE cursorname [ BINARY ] [ INSENSITIVE ] [ SCROLL ]
    CURSOR FOR query
    [ FOR { READ ONLY | UPDATE [ OF column [, ...] ] ]
```

Описание: определяет курсор для доступа к таблице

DELETE

```
DELETE FROM [ ONLY ] table [ WHERE condition ]
```

Описание: удаляет строки из таблицы

DROP AGGREGATE

```
DROP AGGREGATE name type
```

Описание: удаляет определение агрегатной функции

DROP DATABASE

```
DROP DATABASE name
```

Описание: удаляет существующую базу данных

DROP FUNCTION

```
DROP FUNCTION name ( [ type [, ...] ] )
```

Описание: удаляет определенную пользователем функцию C

DROP GROUP

```
DROP GROUP name
```

Описание: удаляет группу

DROP INDEX

```
DROP INDEX index_name [, ...]
```

Описание: удаляет из базы данных существующий индекс

DROP LANGUAGE

```
DROP [ PROCEDURAL ] LANGUAGE 'name'
```

Описание: удаляет определенный пользователем процедурный язык

DROP OPERATOR

```
DROP OPERATOR id ( lefttype | NONE , righttype | NONE )
```

Описание: удаляет из базы данных оператор

DROP RULE

```
DROP RULE name [, ...]
```

Описание: удаляет существующие правила из базы данных

DROP SEQUENCE

```
DROP SEQUENCE name [, ...]
```

Описание: удаляет существующие последовательности из базы данных

DROP TABLE

```
DROP TABLE name [, ...]
```

Описание: удаляет существующие таблицы из базы данных

DROP TRIGGER

```
DROP TRIGGER name ON table
```

Описание: удаляет определение триггера

DROP TYPE

```
DROP TYPE typename [, ...]
```

Описание: удаляет определенные пользователем типы из системного каталога

DROP USER

```
DROP USER name
```

Описание: удаляет пользователя

DROP VIEW

```
DROP VIEW name [, ...]
```

Описание: удаляет из базы данных существующее представление

END

```
END [ WORK | TRANSACTION ]
```

Описание: фиксирует текущую транзакцию

EXPLAIN

```
EXPLAIN [ VERBOSE ] query
```

Описание: показывает план выполнения оператора

FETCH

```
FETCH [ direction ] [ count ] { IN | FROM } cursor
FETCH [ FORWARD | BACKWARD | RELATIVE ] [ # | ALL | NEXT | PRIOR ]
      { IN | FROM } cursor
```

Описание: извлекает строки с помощью курсора

GRANT

```
GRANT privilege [, ...] ON object [, ...]
    TO { PUBLIC | GROUP group | username }
```

Описание: выдает права доступа пользователю, группе пользователей или всем пользователям

INSERT

```
INSERT INTO table [ ( column [, ...] ) ]
    { DEFAULT VALUES | VALUES ( expression [, ...] ) | SELECT query }
```

Описание: вставляет в таблицу новые строки

LISTEN

```
LISTEN name
```

Описание: получение ответа на заданное условие оповещения

LOAD

```
LOAD 'filename'
```

Описание: динамически загружает объектный файл

LOCK

```
LOCK [ TABLE ] name
LOCK [ TABLE ] name IN [ ROW | ACCESS ]
```

```
    { SHARE | EXCLUSIVE } MODE
LOCK [ TABLE ] name IN SHARE ROW EXCLUSIVE MODE
```

Описание: явно блокирует таблицу внутри транзакции

MOVE

```
MOVE [ direction ] [ count ]
    { IN | FROM } cursor
```

Описание: перемещает курсор

NOTIFY

```
NOTIFY name
```

Описание: посылает сигнал всем серверам и клиентам, ожидающим оповещения

REINDEX

```
REINDEX { TABLE | DATABASE | INDEX } name [ FORCE ]
```

Описание: восстанавливает разрушенные системные индексы PostgreSQL в автономном режиме

RESET

```
RESET variable
```

Описание: восстанавливает параметры времени выполнения в их значения по умолчанию

REVOKE

```
REVOKE privilege [, ...]
    ON object [, ...]
    FROM { PUBLIC | GROUP groupname | username }
```

Описание: отменяет права доступа для пользователя, группы пользователей или всех пользователей

ROLLBACK

```
ROLLBACK [ WORK | TRANSACTION ]
```

Описание: откатывает текущую транзакцию

SELECT

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS output_name ] [, ...]
    [ INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
```

```
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT [ ALL ] } select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ FOR UPDATE [ OF tablename [, ...] ] ]
    [ LIMIT { count | ALL } [ { OFFSET | , } start ]]
```

где `from_item` может быть:

```
[ ONLY ] table_name [ * ]
    [ [ AS ] alias [ ( column_alias_list ) ] ]
|
( select )
    [ AS ] alias [ ( column_alias_list ) ]
|
from_item [ NATURAL ] join_type from_item
    [ ON join_condition | USING ( join_column_list ) ]
```

Описание: извлекает строки из таблицы или представления

SELECT INTO

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS output_name ] [, ...]
    INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT [ ALL ] } select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ FOR UPDATE [ OF tablename [, ...] ] ]
    [ LIMIT { count | ALL } [ { OFFSET | , } start ]]
```

где `from_item` может быть:

```
[ ONLY ] table_name [ * ]
    [ [ AS ] alias [ ( column_alias_list ) ] ]
|
( select )
    [ AS ] alias [ ( column_alias_list ) ]
|
from_item [ NATURAL ] join_type from_item
    [ ON join_condition | USING ( join_column_list ) ]
```

Описание: создает новую таблицу из существующей или представления

SET

```
SET variable { TO | = } { value | 'value' | DEFAULT }
SET TIME ZONE { 'timezone' | LOCAL | DEFAULT }
```

Описание: устанавливает параметры времени выполнения

SET CONSTRAINTS

```
SET CONSTRAINTS { ALL | constraint [, ...] } { DEFERRED | IMMEDIATE }
```

Описание: устанавливает режим действия ограничений для текущей транзакции SQL

SET TRANSACTION

```
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL { READ COMMITTED |
SERIALIZABLE }
```

Описание: устанавливает характеристики для текущей транзакции SQL

SHOW

```
SHOW name
```

Описание: показывает параметры времени выполнения

TRUNCATE

```
TRUNCATE [ TABLE ] name
```

Описание: усекает таблицу

UNLISTEN

```
UNLISTEN { notifyname | * }
```

Описание: прекращает прослушивание в ожидании уведомления

UPDATE

```
UPDATE [ ONLY ] table SET col = expression [, ...]
    [ FROM fromlist ]
    [ WHERE condition ]
```

Описание: заменяет значения столбцов таблицы

VACUUM

```
VACUUM [ VERBOSE ] [ ANALYZE ] [ table ]
VACUUM [ VERBOSE ] ANALYZE [ table [ (column [, ...] ) ] ]
```

Описание: очищает и анализирует базу данных PostgreSQL

Этот набор команд получен при помощи утилиты командной строки `psql`. ***В любой момент можно сгенерировать первый список, используя команду*** `\h`, ***и получить более конкретную синтаксическую помощь, введя*** `\h <команда>`.

D

Справочная информация по psql

Параметры командной строки psql

Использование:

```
psql [options] [dbname [username]]
```

Параметры:

`-a`	Отображать весь ввод из сценария
`-A`	Режим вывода таблицы без выравнивания (`-P format=unaligned`)
`-c <query>`	Выполнить только один запрос (query) (или команду, начинающуюся с косой черты) и завершить работу
`-d <dbname>`	Указать имя базы данных (dbname) для установления соединения
`-e`	Отображать запросы, отправленные на сервер
`-E`	Выводить запросы, генерируемые внутренними командами
`-f <filename>`	Выполнить запросы из файла с именем filename и завершить работу
`-F <string>`	Установить разделитель полей (string) (по умолчанию «`|`») (`-P fieldsep=`)
`-h <host>`	Указать компьютер сервера базы данных (host)
`-H`	Режим вывода таблицы HTML (`-P format=html`)
`-l`	Перечислить доступные базы данных и выйти
`-n`	Не использовать readline для ввода
`-o <filename>`	Отправить вывод запроса в файл (`filename`) (или конвейер `|pipe`)

-p <port>	Указать порт сервера базы данных
-P var[=arg]	Установить параметр печати var в arg (см. команду \pset)
-q	Подавление вывода дополнительных сообщений, только вывод результата запроса
-R <string>	Установить разделитель записей (-P recordsep=)
-s	Одношаговый режим (подтверждать каждый запрос)
-S	Однострочный режим (разделитель строк завершает запрос)
-t	Выводить только строки (-P tuples_only)
-T text	Установить параметры тега для таблицы HTML (ширина, рамка) (-P tableattr=)
-U <username>	Указать имя пользователя базы данных
-v name=val	Установить переменную psql с именем name в значение value
-V	Отобразить информацию о версии и завершить работу
-W	Приглашение на ввод пароля (должно выводиться автоматически)
-x	Включить развернутый вывод таблицы (-P expanded)
-X	Пропустить чтение файла инициализации (~/.psqlrc)

Внутренние команды psql

-a	Переключение режимов с выравниванием и без него
\c[onnect] [dbname|-[user]]	Соединение с новой базой данных
\C <title>	Заголовок таблицы
\copy ...	Выполнить SQL-команду COPY на клиентской машине
\copyright	Показать условия использования и распространения PostgreSQL
\d <table>	Описать таблицу (или представление, индекс, последовательность)
\d{t|i|s|v}	Перечислить таблицы/индексы/последовательности/представления
\d{p|S|l}	Перечислить права доступа/системные таблицы/объекты
\da	Перечислить агрегатные функции
\dd [object]	Перечислить комментарии для таблицы, типа, функции или оператора
\df	Перечислить функции
\do	Перечислить операторы
\dT	Перечислить типы данных

(продолжение)

\e [file]	Редактировать буфер текущего запроса или [file] внешним редактором
\echo <text>	Вывести text на стандартный вывод
\f <sep>	Изменить разделитель полей
\h [cmd]	Помощь по синтаксису команд SQL, * – для всех команд
\i <file>	Считывать и выполнять запросы из <file>
\l	Перечислить все базы данных
\lo_export, \lo_import, \lo_list, \lo_unlink	Операции с большими объектами
\o [file]	Послать все результаты запроса в [file] или |pipe
\p	Показать содержимое буфера текущего запроса
\pset <opt>	Задать вывод таблицы <opt> = {format | border | expanded | fieldsep | null | recordsep | tuples_only | title | tableattr | pager}
\q	Выход из psql
\qecho <text>	Записать text в поток вывода запроса (см. \o)
\r	Очистить (сбросить) буфер запроса
\s [file]	Напечатать историю команд или сохранить ее в файле [file]
\set <var> <value>	Установить внутренние переменные
\t	Показывать только строки
\T <tags>	Теги таблицы HTML
\unset <var>	Удалить (сбросить) внутренние переменные
\w <file>	Записать буфер текущего запроса в <file>
\x	Переключение развернутого вывода
\z	Перечислить права доступа к таблице
\! [cmd]	Выйти в оболочку или выполнить команду

Этот набор команд получен из оперативной справки по утилите командной строки **psql.**

E

Схема и таблицы базы данных

Схема базы данных, используемая в примерах этой книги, представляет упрощенную базу данных «клиенты–заказы–товары», изображенную на рис. E.1:

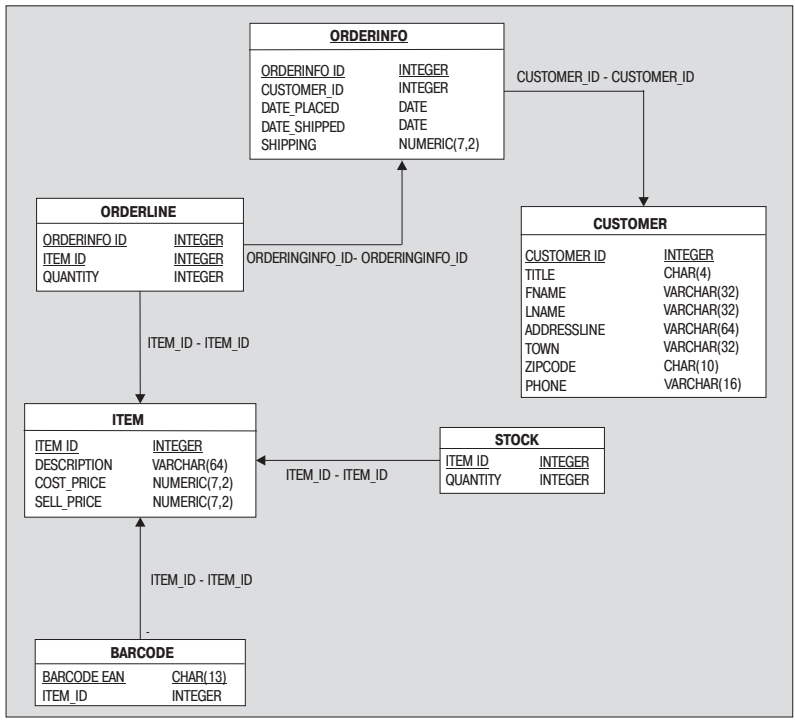


Рис. E.1. Схема базы данных

Таблицы должны создаваться в определенном порядке так, чтобы сначала создавались зависимые таблицы (обусловлено внешними ключами). Такой же порядок необходимо соблюдать и при вставке данных в таблицы. В данном случае он будет таким:

- CUSTOMER
- ORDERINFO
- ITEM
- ORDERLINE
- STOCK
- BARCODE

Приведем SQL, создающий окончательную версию учебной базы данных `bpsimple`, в том числе и внешние ключи:

Таблица customer

```
create table customer
(
    customer_id                    serial,
    title                          char(4),
    fname                          varchar(32),
    lname                          varchar(32) not null,
    addressline                    varchar(64),
    town                           varchar(32),
    zipcode                        char(10) not null,
    phone                          varchar(16),
    CONSTRAINT                     customer_pk PRIMARY KEY(customer_id)
);
```

Таблица orderinfo

```
create table orderinfo
(
    orderinfo_id                   serial,
    customer_id                    integer not null,
    date_placed                    date not null,
    date_shipped                   date,
    shipping                       numeric(7,2) ,
    CONSTRAINT                     orderinfo_pk PRIMARY KEY(orderinfo_id),
    CONSTRAINT orderinfo_customer_id_fk FOREIGN KEY(customer_id) REFERENCES
customer(customer_id)
);
```

Таблица item

```
create table item
(
    item_id                        serial,
```

```
    description                    varchar(64) not null,
    cost_price                     numeric(7,2),
    sell_price                     numeric(7,2),
    CONSTRAINT                     item_pk PRIMARY KEY(item_id)
);
```

Таблица orderline

```
create table orderline
(
    orderinfo_id                   integer not null,
    item_id                        integer not null,
    quantity                       integer not null,
    CONSTRAINT                     orderline_pk PRIMARY KEY(orderinfo_id,
item_id),
   CONSTRAINT orderline_orderinfo_id_fk FOREIGN KEY(orderinfo_id) REFERENCES
orderinfo(orderinfo_id),
    CONSTRAINT orderline_item_id_fk FOREIGN KEY(item_id) REFERENCES
item(item_id)
);
```

Таблица stock

```
create table stock
(
    item_id                        integer not null,
    quantity                       integer  not null,
    CONSTRAINT                     stock_pk PRIMARY KEY(item_id),
    CONSTRAINT stock_item_id_fk FOREIGN KEY(item_id) REFERENCES item(item_id)
);
```

Таблица barcode

```
create table barcode
(
    barcode_ean                    char(13) not null,
    item_id                        integer not null,
    CONSTRAINT                     barcode_pk PRIMARY KEY(barcode_ean),
    CONSTRAINT barcode_item_id_fk FOREIGN KEY(item_id) REFERENCES
item(item_id)
);
```

Загружаемый из Интернета пакет с кодом содержит команды заполнения таблиц для базы данных `bpsimple` в соответствующем порядке.

F

Поддержка больших объектов в PostgreSQL

По традиции базы данных обеспечивали хранение данных в ограниченном количестве форм. Обычно это были числовые значения (целые, с плавающей точкой и фиксированной точкой) и текстовые строки. Часто размер текстовых данных был ограничен. В предыдущих версиях PostgreSQL длина поля была ограничена пределом в несколько тысяч символов.

Теперь же PostgreSQL обеспечивает поддержку большого разнообразия типов данных для столбцов таблицы, в том числе геометрических объектов, сетевых адресов и массивов.

При желании пользователь даже может сам определять типы. Не будем говорить здесь о таких эзотерических типах, информацию о них можно найти в документации по PostgreSQL.

Для нас может быть полезна возможность создать приложение базы данных, которое может обрабатывать произвольные неструктурированные форматы данных, например для картинок. Может возникнуть желание добавить в нашу учебную базу данных фотографии товаров, чтобы веб-интерфейс предоставлял доступ к каталогу в Интернете.

PostgreSQL поддерживает произвольный тип данных, большой двоичный объект или BLOB, подходящий для хранения больших элементов данных. На нем и сосредоточимся.

Добавление картинок в базу данных

Рассмотрим добавление фотографий товаров в нашу базу данных. Есть несколько способов сделать это:

- Путем применения ссылок на файл в файловой системе или в Интернете
- С помощью закодированных длинных текстовых строк
- За счет использования больших бинарных объектов

Первый вариант предлагает вообще избежать хранения картинок в физической базе данных. Идея заключается в том, чтобы поместить все картинки в обычную систему регистрации документов на сервере, который может быть сервером базы данных, файловым сервером или веб-сервером. Сама база данных должна содержать только текстовые ссылки на файл. Любое клиентское приложение должно будет пройти по ссылке, чтобы извлечь картинку.

Необходимо создать дополнительную таблицу в базе данных, чтобы хранить ссылки на картинки. Она очень похожа на таблицу `stock` тем, что для каждого товара предоставляется дополнительная информация:

```
CREATE TABLE image
(
    item_id     INTEGER NOT NULL,
    picture     VARCHAR(512),
    CONSTRAINT  image_pk PRIMARY KEY(item_id),
    CONSTRAINT  image_item_id_fk FOREIGN KEY(item_id) REFERENCES
    item(item_id)
);
```

Добавлены ограничения, преследующие цель обеспечить гарантию возможности добавления картинок только для существующих товаров.

Теперь можно обновить таблицу `image` ссылками на фотографии продуктов:

```
INSERT INTO image VALUES (3, 'http://server/images/rubik.jpg');
INSERT INTO image VALUES (9, '//server/images/coin.bmp');
INSERT INTO image VALUES (5, '/mnt/server/images/frame.png');
```

У такого решения есть свои плюсы и минусы.

Хранение ссылок вместо картинок означает, что размер базы данных остается минимальным, и приложения будут переносимы на другие СУБД, поскольку никакие специальные возможности обработки картинок, известные лишь посвященным, задействованы не были.

Извлечение настоящих картинок также будет очень быстрым, т. к. чтение файла файловой системы обычно выполняется гораздо быстрее запроса к базе данных.

Также существует широкий выбор вариантов размещения картинок. В приведенном выше примере использовались:

- URL для ссылки на веб-сервер
- Ссылка на UNC-файл файловой системы Windows
- Ссылка на NFS UNIX-сервера

Однако посредством ссылок невозможно обеспечить ссылочную целостность базы данных.

Если картинки, хранящиеся в каком-то другом месте, изменяются или удаляются, то база данных не обновляется автоматически. При резервном копировании системы придется уделить внимание как файлам картинок в файловой системе (или где-то еще), так и самой базе данных. К тому же необходимо, чтобы действующие ссылки были представлены в форме, доступной любым клиентам. Например, NFS требует, чтобы все клиенты обращались к разделяемым файлам одним и тем же способом.

В PostgreSQL 7.1 максимальный размер поля увеличен до 1 Гбайт. Для любого практического применения можно сказать, что оно не ограничено. Можно подумать об использовании типа `text` для непосредственного хранения картинок в базе данных. Это возможно, хотя и несколько мудрено.

Вообще-то картинки – это двоичные данные, не очень-то подходящие для символьного типа. Поэтому надо как-то закодировать картинку, вероятно, применив шестнадцатеричную или MIME-кодировку. В результате получатся очень длинные строки, обработка которых также может затруднить соблюдение ограничений, накладываемых клиентскими приложениями, механизмами сетевой передачи или драйверами ODBC. Пространство, необходимое для хранения закодированных строк, также более чем в два раза превышает размер двоичного файла картинки.

Картинки, а на самом деле и любые большие или бинарные объекты, можно хранить в базе данных PostgreSQL при помощи так называемых **больших объектов**, известных еще под именем больших бинарных объектов или BLOB.

BLOB

PostgreSQL поддерживает для столбцов тип `oid` идентификатор объекта, представляет собой ссылку на произвольные данные. Это «блобы», посредством которых можно организовать передачу содержимого любого файла в базу данных и извлечение объекта из базы данных в файл. Следовательно, их можно применять для обработки наших изображений товаров или любых других данных, которые нужно будет сохранить.

Можно изменить определение таблицы image для использования BLOB, указав в качестве типа данных для картинки тип oid:

```
CREATE TABLE image
(
    item_id     INTEGER NOT NULL,
    picture     OID,
    CONSTRAINT  image_pk PRIMARY KEY(item_id),
    CONSTRAINT  image_item_id_fk FOREIGN KEY(item_id) REFERENCES
item(item_id)
);
```

Импорт и экспорт

PostgreSQL предоставляет несколько функций, которые можно использовать в запросах SQL для вставки данных BLOB в таблицу, извлечения их оттуда и удаления.

Чтобы вставить картинку в таблицу, можно применить функцию SQL lo_import следующим образом:

```
INSERT INTO image VALUES (3, lo_import('/tmp/image.jpg'));
```

Содержимое указанного файла считывается в BLOB и сохраняется в базе данных. В таблице image теперь появится oid, не равный NULL, который будет ссылаться на BLOB:

```
bpfinal=# SELECT * FROM image;
item_id | picture
--------+----------
3 |  163055
(1 row)

bpfinal=>
```

Можно увидеть все большие объекты, хранящиеся в базе данных, выполнив внутреннюю команду psql \lo_list или \dl:

```
bpfinal=# \dl
    Large objects
   ID   | Description
--------+-------------
 163055 |
(1 row)

bpfinal=>
```

Большие объекты извлекаются из базы данных командой lo_export, которая записывает файл, в котором хранится содержимое BLOB:

```
bpfinal=# SELECT lo_export(picture, '/tmp/image2.jpg')
bpfinal-# FROM image WHERE item_id = 3;
 lo_export
-------------
```

```
         1
(1 row)

bpfinal=#
```

Можно удалить большой объект из базы данных, используя `lo_unlink`:

```
bpfinal=# SELECT lo_unlink(picture) FROM image WHERE item_id = 3;
 lo_unlink
------------
         1
(1 row)

bpfinal=# \dl
  Large objects
 ID | Description
----+-------------
(0 rows)

bpfinal=#
```

При удалении больших объектов необходимо соблюдать осторожность, т. к. все ссылки на них останутся:

```
bpfinal=# SELECT * FROM image;
 item_id | picture
---------+----------
       3 |  163055
(1 row)

bpfinal=#
```

Поскольку операции над большими объектами и их идентификаторами объектов, по существу, не связаны между собой, работая с BLOB, необходимо заботиться о выполнении и тех и других. Так, удаляя BLOB, следует установить ссылку на объект в `NULL`, чтобы избежать ошибок в клиентских приложениях:

```
bpfinal=# UPDATE image SET picture=null WHERE item_id = 3;
```

В рассмотренных примерах для манипулирования большими объектами применялась утилита `psql`. Важно понимать, что функции импорта и экспорта `lo_import` и `lo_export` выполняются сервером базы данных, а не `psql`. Любое клиентское приложение, использующее операторы SQL, может применять эти функции для добавления и обновления больших объектов.

Но все же необходимы три предупреждения.

Первое заключается в том, что поскольку импорт выполняется сервером, то файлы, считываемые и записываемые функциями `import` и `export`, должны задаваться так, чтобы их путь и имя файла были доступны серверу, а не клиенту. Если в `psql` просто написать:

```
INSERT INTO image VALUES (3, lo_import('image.jpg'));
```

и надеяться, что PostgreSQL вставит содержимое файла в текущий каталог, то будет выведено сообщение об ошибке. Дело в том, что функция импорта не может найти файл. Необходимо разместить (временно) файлы для импорта в такое место, где они будут доступны серверу. Аналогично необходимо использовать полные имена файлов для экспорта бинарных объектов.

Второй момент, на который следует обратить внимание, заключается в том, что экспортируемые файлы должны быть помещены в каталог, в который пользователь сервера может записывать данные, то есть туда, где пользователь `postgres` имеет право на создание файлов.

И третье – все операции с большими объектами должны происходить внутри транзакции SQL, то есть между операторами `BEGIN` и `COMMIT` или `END`. По умолчанию `psql` выполняет каждый оператор SQL в его собственной транзакции, поэтому это не представляет особой проблемы, но клиентское приложение, осуществляющее импорт и экспорт, должно быть написано с применением транзакций.

Удаленный импорт и экспорт

Функции SQL `lo_import` и `lo_export` используют файловую систему сервера, что может быть неудобно при создании BLOB. Тем не менее `psql` содержит внутренние команды, обеспечивающие экспортирование и импортирование бинарных объектов с удаленных клиентских машин.

Можно добавить BLOB в базы данных командой `\lo_import`, которой передано локальное имя файла. В этом случае файлы текущего каталога вполне подходят, и можно вывести список объектов с помощью команды `\lo_list`:

```
bpfinal=# \lo_import image.jpg
lo_import 163059
bpfinal=# \lo_list
    Large objects
   ID   | Description
--------+-------------
 163059 |
(1 row)

bpfinal=#
```

Теперь надо сопоставить BLOB таблице `image`, обновив соответствующую строку:

```
bpfinal=# UPDATE image SET picture=163059 WHERE item_id = 3;
UPDATE 1
bpfinal=# SELECT * FROM image;
 item_id | picture
---------+----------
       3 |  163059
(1 row)

bpfinal=#
```

Можно извлечь BLOB с помощью \lo_export, указав необходимый идентификатор объекта и файл для записи. Опять-таки достаточно локального имени файла:

```
bpfinal=# \lo_export 163059 image2.jpg
lo_export

bpfinal=#
```

Наконец, можно удалить большой объект командой \lo_unlink:

```
bpfinal=# \lo_unlink 163059
lo_unlink 163059

bpfinal=#
```

Программирование больших объектов

Как и следовало предполагать, есть возможность использовать функции импорта и экспорта BLOB из языков программирования, поддерживаемых PostgreSQL.

Из C с помощью библиотеки libpq можно вызывать функции lo_import, lo_export и lo_unlink практически так же, как и ранее:

```
Oid lo_import(PGconn *conn, const char *filename);
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
int lo_unlink(PGconn *conn, Oid lobjId);
```

Приведем пример программы, импортирующей файл с картинкой в базу данных. Обратите внимание, что функции обработки больших объектов должны вызываться внутри транзакции:

```
#include <stdlib.h>
#include <libpq-fe.h>

int main()
{
  PGconn *myconnection = PQconnectdb("");
  PGresult *res;
  Oid blob;

  IF(PQstatus(myconnection) == CONNECTION_OK)
    printf("connection made\n");
  ELSE
    printf("connection failed\n");

  res = PQexec(myconnection, "begin");
  PQclear(res);

  blob = lo_import(myconnection, "image.jpg");
  printf("import returned oid %d\n", blob);
```

```
    res = PQexec(myconnection, "end");
    PQclear(res);

    PQfinish(myconnection);
    return EXIT_SUCCESS;
}
```

Скомпилировав и запустив программу, увидим сообщение о новом идентификаторе бинарного объекта.

Более подробная информация о компилировании и запуске программ `libpq` приведена в главе 13:

```
$ make import
cc -I/usr/local/pgsql/include  -L/usr/local/pgsql/lib -lpq  import.c   -o
import
$ PGDATABASE=bpfinal ./import
connection made
import returned oid 163066
$
```

Из других языков большие объекты могут импортироваться и экспортироваться аналогичным образом. Например, интерфейс Tcl к PostgreSQL содержит функции `pg_lo_import`, `pg_lo_export` и `pg_lo_unlink`.

Для более тонкого контроля за доступом к большим объектам PostgreSQL предусматривает ряд низкоуровневых функций, родственных открытию, чтению, записи и другим им подобным, для обычных файлов:

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
int lo_close(PGconn *conn, int fd);
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
int lo_write(PGconn *conn, int fd, char *buf, size_t len);
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
Oid lo_creat(PGconn *conn, int mode);
int lo_tell(PGconn *conn, int fd);
```

За подробной информацией об этих функциях обращайтесь к документации.

Алфавитный указатель

Символы

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

A

Б

В

Г

Д

Ж

З

И

К

Л

М

Н

О

П

Р

C

Т

У

Ф

Х

Ц

Ч

Ш

Э

Я