Министерство образования и науки Российской Федерации
Сибирский государственный аэрокосмический университет
имени академика М. Ф. Решетнева

Е. П. Моргунов, О. Н. Моргунова

АДМИНИСТРИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ

*Утверждено редакционно-издательским советом университета
в качестве учебного пособия для бакалавров, обучающихся
по направлениям подготовки 230400.62 «Информационные системы
и технологии» и 230700.62 «Прикладная информатика»
всех форм обучения*

Красноярск 2015

УДК 004.7
ББК 32.973.202-018.2
М79

Рецензенты:
кандидат технических наук, главный специалист-эксперт В. А. Морозов
(Управление Федерального казначейства по Красноярскому краю);
кандидат технических наук, доцент А. Н. Горошкин
(Сибирский государственный аэрокосмический университет
имени академика М. Ф. Решетнева)

Моргунов, Е. П.
М79 Администрирование информационных систем : учеб. пособие / Е. П. Моргунов, О. Н. Моргунова ; Сиб. гос. аэрокосмич. ун-т. – Красноярск, 2015. – 90 с.

Описаны основные процедуры, выполняемые администратором информационной системы. Представлены базовые технологии, которыми должен владеть администратор. Даются конкретные рекомендации по выполнению практических задач.

Предназначено для студентов, обучающихся по направлениям подготовки 230400.62 «Информационные системы и технологии» и 230700.62 «Прикладная информатика». Может быть полезно широкому кругу студентов, желающих ознакомиться с основами администрирования информационных систем.

ББК 32.973.202-018.2
УДК 004.7

Учебно-теоретическое издание

Моргунов Евгений Павлович
Моргунова Ольга Николаевна

АДМИНИСТРИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ

Учебное пособие

Редактор *О. А. Кругликова*
Оригинал-макет и верстка *А. А. Ловчиковой*

Подписано в печать 26.01.2015. Формат 60×84/16. Бумага офсетная.
Печать плоская. Усл. п. л. 5,7. Уч.-изд. л. 6,1. Тираж 55 экз.
Заказ . С 20/15.

Санитарно-эпидемиологическое заключение
№ 24.04.953.П.000032.01.03. от 29.01.2003 г.

Редакционно-издательский отдел Сиб. гос. аэрокосмич. ун-та.
Опечатано в отделе копировально-множительной техники
Сиб. гос. аэрокосмич. ун-та.
660014, г. Красноярск, просп. им. газ. «Красноярский рабочий», 31.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Основой информационной системы, как правило, является база данных, для управления которой используется система управления базами данных (СУБД). Поэтому администрирование информационных систем неотделимо от администрирования СУБД. Функции администратора СУБД разнообразны. Конечно, первое, что должен уметь такой специалист, это установить сервер баз данных и выполнить его первоначальную настройку. Другим важным компонентом квалификации администратора является хорошее знание языка SQL, в том числе умение оптимизировать запросы к базе данных с точки зрения быстроты их выполнения. В составе СУБД для управления сервером баз данных предусмотрены различные механизмы, в том числе конфигурационные файлы, команды языка SQL и системные таблицы. Администратор должен хорошо представлять себе эти механизмы. Он отвечает за работу сервера баз данных, но на этом сервере хранятся не только данные пользователей, но и так называемые хранимые процедуры. Эти процедуры разрабатываются на различных языках и используются в различных ситуациях для повышения производительности работы прикладных программ. Для обеспечения целостности и согласованности данных используются триггеры, которые срабатывают при наступлении тех или иных событий в базе данных. Администратор должен хорошо понимать принципы их работы с тем, чтобы при необходимости оказать помощь прикладному программисту. Одним из важнейших механизмов организации одновременной работы многих пользователей с одной базой данных являются транзакции. Без понимания принципов их функционирования администратор не сможет разрешать конфликты, которые могут возникать между приложениями, использующими одни и те же таблицы в базе данных. А еще очень важным является умение администратора создать учетные записи пользователей и предоставить им сбалансированные привилегии доступа к ресурсам базы данных. Все эти вопросы рассматриваются в учебном пособии.

В настоящее время существует целый ряд СУБД, широко применяемых для решения практических задач. В этом ряду есть как коммерческие, так и свободные программные продукты (free software), такие как MySQL и PostgreSQL. У каждой из этих СУБД есть свои достоинства и свои приверженцы, но в силу ограниченности объема нашего учебного курса мы выбрали для изучения только одну из них, а именно PostgreSQL. Важно отметить, что

в качестве операционной системы (ОС) нами выбрана ОС UNIX, а точнее, Linux Debian и FreeBSD.

Это учебное пособие предназначено для получения практических навыков администрирования. Текст пособия написан таким образом, что многие важные знания студент должен получить в результате выполнения заданий, представленных в конце каждой главы. В основном тексте эти знания не представлены. Предполагается, что значительная часть этих заданий будет выполняться самостоятельно с помощью технической документации на СУБД PostgreSQL. В описаниях заданий зачастую даются и указания к их выполнению.

В пособии используются различные виды шрифтов для выделения различных фрагментов текста в зависимости от их назначения. Команды, вводимые пользователем, выделяются полужирным шрифтом, например:

```
psql –d test –U root
```

Если вся команда не умещается на одной строке текста, то она переносится на следующие строки, но вы должны вводить ее на одной строке, например:

```
EXPLAIN SELECT * FROM students WHERE mark_book >= 10000 AND
mark_book <= 10100;
```

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

```
EXPLAIN SELECT * FROM students;
```

на экран будет выведено следующее:

```
                        QUERY PLAN
-----------------------------------------------------------
Seq Scan on students  (cost=0.00..7.20 rows=320 width=69)
(1 строка)
```

Тексты программ, которые приведены в учебном пособии, напечатаны также моноширинным шрифтом. Например:

```
CREATE FUNCTION add_numbers( x integer, y integer ) RETURNS
integer AS $$
  SELECT x + y;
$$ LANGUAGE SQL;
```

В пособии часто будут даваться указания ввести ту или иную команду. Это означает, что вам следует не только набрать на клавиатуре текст этой команды, но также нажать клавишу Enter для ее выполнения. Для сокращения объема текста мы не будем повторять слова «Нажмите клавишу Enter» каждый раз.

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

1. СОЗДАНИЕ РАБОЧЕЙ СРЕДЫ

1.1. Основные понятия баз данных

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

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

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

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

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

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

В настоящее время преобладают базы данных реляционного типа. Их характерной чертой является тот факт, что данные воспринимаются пользователем как таблицы. В распоряжении пользователя имеются операторы для выборки данных из таблиц, а также для вставки новых данных, обновления и удаления имеющихся данных.

1.2. Установка систем управления базами данных и первоначальная настройка

Устанавливать рекомендуется последнюю **стабильную** версию СУБД. На данном «историческом» отрезке времени она имеет номер 9.x.x. Здесь символами «x» обозначены младшие номера версий, которые подвержены более частым изменениям, чем старший номер версии.

Установить СУБД можно как из заранее скомпилированных (бинарных) пакетов, так и из исходных текстов. Для установки из бинарных пакетов в среде операционной системы Debian следует воспользоваться командой

```
apt-get install postgresql-9.3
```

Эта команда позволит установить только непосредственно сервер СУБД, а дополнительные подсистемы нужно устанавливать отдельно. Например, команда для установки клиентских утилит такова:

```
apt-get install postgresql-client-9.3
```

В среде ОС FreeBSD существует так называемая коллекция портов (ports and packages collection). Для установки СУБД можно воспользоваться ее возможностями. Подробнее об установке СУБД из бинарных пакетов, как в ОС Debian, так и FreeBSD, смотрите на сайте http://www.postgresql.org в разделе Download.

Теперь мы подробно рассмотрим процедуру установки СУБД PostgreSQL из исходных текстов в среде операционной системы UNIX, а точнее, Linux Debian и FreeBSD. Загрузить исходные тексты можно с сайта http://www.postgresql.org. На этом сайте пройдите по ссылке «Download» и на этой странице в разделе Source code откройте ссылку «file browser», а в открывшемся списке каталогов выберите тот, который соответствует самой последней версии СУБД. Войдите в этот каталог и в нем выберите для загрузки последнюю **стабильную** версию архивного файла с именем вида «postgresql-9.x.x.tar.gz» или «postgresql-9.x.x.tar.bz2».

Подготовка к установке СУБД заключается в следующем. Зарегистрируйтесь в системе под именем root. Скопируйте архивный файл в каталог /usr/src (можно выбрать другой каталог), извлеките файлы из архива, а затем перейдите во вновь созданный подкаталог postgresql-9.x.x:

```
cp postgresql-9.x.x.tar.gz /usr/src
cd /usr/src
tar xzvf postgresql-9.x.x.tar.gz
cd postgresql-9.x.x
```

либо

```
cp postgresql-9.x.x.tar.bz2 /usr/src
cd /usr/src
bzcat postgresql-9.x.x.tar.bz2 | tar xvf -
cd postgresql-9.x.x
```

Обратите внимание, что в команде извлечения файлов из архива последний символ – дефис. Он означает, что команда tar читает данные не из файла, а со стандартного ввода.

Процедура установки состоит из ряда шагов, которые мы будем нумеровать для придания всей процедуре большей четкости.

1. Первым шагом является конфигурирование иерархии исходных текстов СУБД. Эта задача решается традиционным для свободного (поставляемого в исходных текстах) программного обеспечения способом, а именно, путем запуска программы configure. Она может получать различные параметры в зависимости от требований, предъявляемых к устанавливаемому программному продукту, и с учетом настроек операционной системы. Программа (скрипт) configure написана на языке shell. Однако подобные скрипты не пишутся вручную, а создаются с помощью специальных программ автоматического конфигурирования.

Список всех параметров, принимаемых программой configure, можно получить с помощью команды (обратите внимание на два дефиса перед параметром help)

```
/configure --help
```

Из множества параметров мы опишем лишь некоторые, которые представляют, на наш взгляд, наибольший интерес. Параметр --enable-nls позволяет всем программам, входящим в комплект СУБД, выводить сообщения на том языке, который установлен параметрами

локализации операционной системы. Например, в ОС Debian это будет, скорее всего, «ru_RU.UTF-8», а в ОС FreeBSD – «ru_RU.KOI8-R». Сообщения будут выводиться на русском языке. Посмотреть эти параметры можно в обеих системах с помощью команды

```
locale
```

СУБД PostgreSQL позволяет разрабатывать хранимые процедуры на языке Perl. Для реализации этой возможности служит параметр --with-perl.

Если у вас возникнет желание изучить работу СУБД на уровне исходных текстов, то мы рекомендуем вам скомпилировать их с поддержкой отладчика и отключенной оптимизацией объектного кода. Первая задача решается путем использования параметра --enable-debug (обратите внимание на два дефиса перед этим параметром), а вторая – путем назначения переменной среды операционной системы CFLAGS значения -O0, которое предписывает компилятору генерировать неоптимизированный код. Изучать в отладчике оптимизированный код гораздо сложнее.

Все программы СУБД PostgreeSQL будут установлены в каталог /usr/local/pgsql.

Для выполнения поставленной задачи введите команду (она вводится на одной строке):

```
./configure --enable-nls --enable-debug CFLAGS=-O0 >
conf_log.txt 2>&1 &
```

Все сообщения, выводимые в процессе конфигурирования, мы перенаправим в файл-журнал conf_log.txt. В него будут поступать как сообщения, направляемые на стандартный вывод, так и сообщения об ошибках. Этим управляет компонент 2>&1 (обратите внимание, что в этом компоненте нет пробелов). Последний знак «&» в командной строке означает, что команда будет выполняться в фоновом режиме. Поэтому вы можете наблюдать процесс с помощью команды

```
tail –f conf_log.txt
```

В любой момент вы можете прервать этот просмотр нажатием клавишей Ctrl-C, при этом процесс будет продолжаться. Конечно, вы можете запустить эту команду и в обычном режиме, а не как фоновый процесс, просто не вводя знак «&» в самом конце команды.

По окончании процедуры конфигурирования загляните в файл conf_log.txt. Если в конце файла нет сообщения об ошибке, то все в порядке. Кроме того, программа configure сама создает файлы-журналы в текущем каталоге: config.log и config.status. Однако они не являются точной копией файла conf_log.txt.

Процесс конфигурирования может завершиться с ошибкой. В этом случае сообщение об ошибке вы увидите в самом конце файла conf_log.txt. Если оно гласит, что не найдена какая-нибудь библиотека, тогда нужно приостановить процесс установки PostgreSQL и разобраться, почему программа конфигурирования не нашла библиотеку. Возможны две причины: библиотека в вашей операционной систему установлена, но находится в нестандартном месте, а вторая причина – библиотека не установлена вообще. В первом случае может помочь использование параметра --with-libraries, а во втором случае придется сначала установить эту недостающую библиотеку, а потом повторить команду configure. Вообще, если программа configure не смогла найти требуемые ей файлы, посмотрите с помощью команды

```
./configure --help
```

нет ли параметра, отвечающего за эти файлы.

В ОС Debian, вам, вероятно, придется установить библиотеки readline, zlib и gettext. Можно использовать такие команды:

```
apt-get install libreadline6-dev
apt-get install zlib1g-dev
apt-get install gettext
```

В ОС FreeBSD эти библиотеки, как правило, уже установлены.

2. Следующий этап – компиляция программ. Для этой цели нужно использовать только утилиту GNU make. Если она не установлена в вашей операционной системе, то установите ее, загрузив исходные тексты с сайта http://www.gnu.org. В операционной системе Debian по умолчанию используется именно такая программа make. А в ОС FreeBSD ее нужно установить. Она устанавливается в каталог /usr/local/bin. Поскольку «родная» утилита make находится в каталоге /usr/bin, то утилиту GNU make в каталоге /usr/local/bin нужно для удобства использования переименовать в gmake.

Команда для компиляции исходных текстов в среде ОС Debian будет такой:

```
make world > make_log.txt 2>&1 &
```

Для слежения за ходом процесса используйте команду

```
tail -f make_log.txt
```

В команде для ОС FreeBSD нужно make заменить на gmake.

Процесс может занять от двух-трех минут до получаса, в зависимости от производительности вашего компьютера. Последняя строка, выведенная в файл-журнал, должна быть такой:

```
PostgreSQL, contrib, and documentation successfully made.
Ready to install.
```

3. Теперь нужно установить скомпилированные программы в системные каталоги операционной системы. Команда для ОС Debian будет такой:

```
make install-world > make_install_log.txt 2>&1 &
```

Для слежения за ходом процесса используйте команду

```
tail -f make_install_log.txt
```

В команде для ОС FreeBSD нужно make заменить на gmake.

4. Для удобства использования утилит, входящих в комплект СУБД, рекомендуется добавить путь к этим утилитам в переменную среды PATH. При использовании командного процессора Bourne shell (/bin/sh) или другого, совместного с ним (например, /bin/bash), нужно поступить так: если вы хотите, чтобы изменения были применены для всех пользователей системы, то корректировать нужно файл /etc/profile, а если необходимо сделать изменения только для конкретного пользователя, то нужно корректировать файл ~/.profile в домашнем каталоге пользователя (знак ~ означает домашний каталог пользователя). Перед строкой

```
export PATH
```

нужно добавить строку

```
PATH=/usr/local/pgsql/bin:$PATH
```

Если вы решили корректировать файл ~/.profile в домашнем каталоге пользователя, а в нем вообще нет команды export PATH, тогда добавьте туда обе команды:

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

Если вы используете csh or tcsh, тогда команда будет такой (изменения нужно внести в файл ~/.cshrc):

```
set path = ( /usr/local/pgsql/bin $path )
```

Примечание. При установке операционной системы FreeBSD могут устанавливаться и утилиты СУБД PostgreSQL, но они имеют более старую версию. Поэтому если вы не выполнили рекомендуемой корректировки переменной PATH, то при запуске утилит без указания полного пути к ним вы получите доступ к их более старым версиям.

Аналогично можно настроить и путь к электронным руководствам по утилитам СУБД. В те же файлы нужно добавить строки:

```
MANPATH=:/usr/local/pgsql/man
export MANPATH
```

Двоеточие означает, что содержимое переменной MANPATH будет добавлено в конец списка путей, определяемых в файле /etc/manpath.config.

Если вы вносили изменения в файлы /etc/profile, то необходимо перезагрузить операционную систему, чтобы эти изменения вступили в силу.

5. Для запуска сервера СУБД PostgreSQL необходимо наличие специальной учетной записи в операционной системе. Как правило, такой пользователь имеет имя postgres. Рекомендуем вам также использовать это имя. Создайте учетную запись с помощью команды adduser. Основную часть параметров можно принимать по умолчанию. Если у вас нет особых причин сделать иначе, то выберите в качестве командного интерпретатора Bourne shell (sh) из списка, предложенного утилитой adduser.

6. Теперь необходимо инициализировать кластер баз данных. Создайте каталог, в котором будет располагаться база данных. Затем измените владельца этого каталога на пользователя postgres:

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

Войдите в систему под именем postgres или, используя команду su, «притворитесь» этим пользователем (обратите внимание на пробелы слева и справа от дефиса):

```
su - postgres
```

Инициализация кластера баз данных выполняется так:

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

На экран выводится целый ряд сообщений. Обратите внимание на информацию о параметрах локализации. Например, в ОС FreeBSD будет выведено примерно такое сообщение:

```
The database cluster will be initialized with locale
ru_RU.KOI8-R.
The default database encoding has accordingly been set to
KOI8.
```

Последние сообщения примерно такие:

```
Success. You can now start the database server using:

    /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
or
    /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l
logfile start
```

7. СУБД успешно установлена, значит, можно попытаться запустить серверный процесс.

Примечание. Это выполняется только от имени пользователя postgres (т. е. суперпользователя СУБД), а не от имени пользователя root.

Войдите в систему под именем пользователя postgres и выполните команду (она вводится на одной строке)

```
/usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data
-l logfile start
```

На экран будет выведено сообщение

```
server starting
```

В ОС Debian можно проверить, запущен ли серверный процесс, с помощью команды

```
ps axw
```

В ОС FreeBSD требуется добавить дефис:

```
ps -axw
```

На экране вы найдете примерно такую строку:

```
37561  v0  I      0:00,06 /usr/local/pgsql/bin/postgres -D
/usr/local/pgsql/data
```

При первом запуске серверного процесса создается файл-журнал logfile в каталоге /home/postgres.

Если вы все сделали правильно, то при помощи команды

ps –axw

вы можете увидеть, что сервер успешно запущен:

```
779  ??  Ss     0:00,03 postgres: writer process    (postgres)
780  ??  Ss     0:00,00 postgres: stats collector process
(postgres)
651 con- I      0:00,23 /usr/local/pgsql/bin/postgres -D
/usr/local/pgsql/data
```

8. Если сервер СУБД запущен, можно создать базу данных. В качестве ее имени выберем test. От имени пользователя postgres выполните команду

/usr/local/pgsql/bin/createdb test

9. Пробуем подключиться к только что созданной базе данных test (опять как пользователь postgres):

/usr/local/pgsql/bin/psql test

На экране вы увидите:

```
psql (9.2.4)
Введите "help", чтобы получить справку.

test=#
```

Для выхода введите команду

\q

10. Для останова сервера баз данных необходимо выполнить команду:

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

На этом установка СУБД PostgreSQL может считаться в основном завершенной. Добавим лишь несколько замечаний.

Конечно, настоящее краткое руководство не может и не должно заменять документацию, входящую в комплект поставки СУБД. Традиционно мы рекомендуем обратиться к электронным руководствам man. Только не забывайте, что при установке операционной системы FreeBSD устанавливаются и утилиты PostgreSQL более ранних версий, а с ними устанавливаются и man-страницы. Поэтому для получения доступа к документации, установленной в каталог /usr/local/pgsql/man, вы можете поступить так:

```
cd /usr/local/pgsql/share/man
man -M . psql
```

или так:

```
man -M /usr/local/pgsql/share/man psql
```

Вместо psql указывайте имя одной из интересующих вас утилит, которые собраны в каталоге /usr/local/pgsql/bin.

В комплект поставки входит также и очень подробная документация в формате HTML. Она находится в каталоге /usr/local/pgsql/share/doc/html. Стартовый файл, содержащий оглавление, – index.html.

1.3. Запуск и останов сервера

Если вы хотите, чтобы сервер СУБД PostgreSQL автоматически запускался при загрузке операционной системы и выгружался при ее выключении, то в ОС FreeBSD используйте файлы /etc/rc.local и /etc/rc.shutdown.local (если таких файлов еще нет, создайте их). Для выполнения операций с этими файлами необходимо зарегистрироваться в системе под именем пользователя root.

В файл /etc/rc.local добавьте команду

```
su - postgres -c '/usr/local/pgsql/bin/pg_ctl -D
/usr/local/pgsql/data -l logfile start'
```

Эту команду можно ввести в одну строку или использовать символ «/» для продолжения команды на следующей строке, аналогично тому, как в программах на языке С «склеиваются» строковые константы.

В файл /etc/rc.shutdown.local добавьте команду

```
su - postgres -c '/usr/local/pgsql/bin/pg_ctl -D
/usr/local/pgsql/data stop'
```

Обе команды содержат следующий компонент:

```
su – postgres
```

Он указывает операционной системе, что выполнять команду, заключенную в одинарные кавычки, необходимо от имени пользователя postgres. При этом обратите внимание на наличие пробелов слева и справа от дефиса.

Можно предложить еще один способ запуска и останова сервера баз данных. Для этого в каталоге /home/postgres нужно создать файл pg_start, содержащий следующую команду для запуска сервера БД:

```
/usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l log-
file start
```

Назначьте этому файлу права на исполнение:

```
chmod 755 pg_start
```

Теперь при необходимости запустить сервер баз данных нужно из учетной записи postgres выполнить этот файл:

```
./pg_start
```

Для останова сервера баз данных необходимо в каталоге /home/postgres создать файл pg_stop, поместив в него одну строку:

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

Назначьте этому файлу права на исполнение:

```
chmod 755 pg_stop
```

Перед завершением работы операционной системы нужно запускать этот файл для корректного «выключения» сервера баз данных:

```
./pg_stop
```

1.4. Программа psql – интерактивный терминал PostgreSQL

Для доступа к серверу баз данных в комплект PostgreSQL входит интерактивный терминал psql. Мы его уже упоминали выше. Сейчас мы опишем возможности psql более подробно.

Для получения краткой справки по командам этой утилиты нужно в ответ на приглашение ввести

```
\?
```

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

```
\dt
```

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

```
\d   students
```

Для получения списка всех SQL-команд нужно выполнить команду

```
\h
```

Для вывода описания конкретной SQL-команды введите, например:

```
\h  CREATE TABLE
```

Эта утилита позволяет сокращать объем ручного ввода за счет дополнения вводимой команды «силами» psql. Например, при вводе SQL-команды можно использовать клавишу Tab для дополнения вводимого ключевого слова команды или имени таблицы базы данных. Например, при вводе команды CREATE TABLE можно, введя символы «cr», нажать клавишу Tab – psql дополнит это слово до «create». Аналогично можно поступить и со словом TABLE. Для его ввода достаточно ввести лишь буквы «ta» и нажать клавишу Tab. Если вы

ввели слишком мало букв для того, чтобы psql мог однозначно идентифицировать ключевое слово, дополнения не произойдет. Но в таком случае вы можете нажать клавишу Tab дважды и получить список всех ключевых слов, начинающихся с введенной вами комбинации букв.

1.5. Создание учетной записи пользователя базы данных

Важно помнить, что учетная запись пользователя в операционной системе и учетная запись пользователя в СУБД – это разные вещи, хотя их имена могут совпадать, как это имеет место для пользователя postgres. Пользователь root, имеющий особое значение с точки зрения операционной системы, с точки зрения СУБД PostgreSQL такого значения не имеет. Поэтому, если вы хотите, чтобы в рамках СУБД была учетная запись с именем root, то ее нужно создать с помощью SQL-команды CREATE USER. Для этого войдите в операционную систему как пользователь postgres и подключитесь к базе данных test с помощью команды

```
/usr/local/pgsql/bin/psql test
```

Если вы добавите путь /usr/local/pgsql/bin в переменную PATH для пользователя postgres, как это было рекомендовано выше, то команда будет более короткой:

```
psql test
```

Получив доступ к базе данных, в ответ на приглашение

```
test=#
```

выполните команду (не забудьте поставить точку с запятой в конце команды)

```
CREATE USER root;
```

При успешном выполнении этой команды СУБД «ответит» вам:

```
CREATE ROLE
```

Выйдите из программы psql и попробуйте снова подключиться к базе данных test, но уже как пользователь root:

```
psql –d test –U root
```

Если вы не указали имя пользователя с помощью параметра -U, то по умолчанию используется то же имя, которое имеет текущий пользователь операционной системы. Однако если в СУБД нет пользователя с таким именем, вы получите сообщение об ошибке.

Рекомендуем вам поэкспериментировать с утилитой psql, подключаясь к базе данных test от имени разных пользователей СУБД, регистрируясь в операционной системе также под разными именами.

Создать учетную запись пользователя можно и с помощью утилиты, запускаемой непосредственно из командной строки операционной системы. Это делается так:

```
/usr/local/pgsql/bin/createuser  new_user
```

Для получения справки по этой утилите следует сделать так (обратите внимание на два дефиса перед словом «help»):

```
/usr/local/pgsql/bin/createuser  --help
```

1.6. Установка системы управления базами данных PostgreSQL в среде оперативной системы Windows

Инсталляционный архивный файл для ОС Windows также можно загрузить с сайта http://www.postgresql.org.

Установите СУБД в каталог, путь к которому не содержит пробелов, например C:\Pg или C:\pgsql. Не устанавливайте СУБД в традиционный каталог C:\Program Files, так как путь к нему содержит пробельный символ. Это требование важно потому, что впоследствии вы можете пожелать установить, например, среду разработки MSYS или MinGW, а многие утилиты, входящие в комплект MSYS и MinGW, были «рождены» в среде UNIX и «не любят» пробелы в именах каталогов.

При запуске утилиты psql в среде Windows нужно сделать следующее. Выберите строку CommandPrompt из списка доступных опций в группе PostgreSQL 9.x, находящейся в списке «Все программы» (кнопка «Пуск»). Откроется окно с командной строкой. Сначала необходимо изменить свойства этого окна, а именно: выбрать шрифт Lucida Console. Затем уже в окне введите команду

```
chcp 1251
```

Эта команда назначает текущую кодовую страницу CP1251. Теперь запустите утилиту psql:

```
psql –d postgres –U postgres
```

Если при установке СУБД вы назначили пароль для пользователя postgres, то вас попросят ввести его при запуске psql. Теперь вы можете ввести команду

```
\?
```

и получить подсказку на русском языке.

Кроме скромной утилиты psql в комплект СУБД входит мощная утилита pgAdminIII, имеющая графический интерфейс пользователя. В частности, она имеет специальный модуль (интерфейс) для ввода SQL-команд (меню Tools → Query tool). Рекомендуем вам самостоятельно изучить возможности утилиты pgAdminIII с помощью документации и метода проб и ошибок.

Контрольные вопросы и задания

1. Для чего предназначена утилита configure?
2. Каким образом можно перенаправить вывод команды configure в файл-журнал?
3. Выполните всю процедуру установки СУБД PostgreSQL в среде операционной системы Linux (например, Debian) или FreeBSD.
4. Организуйте автоматический запуск и останов сервера PostgreSQL при старте и останове операционной системы. Для этого внесите необходимые изменения в соответствующие системные файлы.
5. Создайте командные файлы для ручного запуска и останова сервера СУБД. Не забудьте назначить этим файлам соответствующие привилегии доступа, чтобы файлы стали исполняемыми.
6. Ознакомьтесь с утилитой psql с помощью встроенной справки, а также с помощью справки, вызываемой по команде

```
/usr/local/pgsql/bin/psql  --help
```

7. Самостоятельно установите программу pgAdmin и изучите основные приемы работы с ней.
8. Самостоятельно установите программу phpPgAdmin и изучите основные приемы работы с ней.
9. Иногда сервер баз данных PostgreSQL не запускается, а на экран выводится сообщение об ошибке. С помощью команды

```
man postgres
```

самостоятельно ознакомьтесь с электронным руководством по утилите postgres. В разделе «Diagnostics (диагностика)» изучите причины возможного отказа сервера от запуска.

2. ЯЗЫК SQL

2.1. Основные понятия реляционной модели

Рассмотрим простую систему, в которой всего две таблицы: «Студенты» и «Успеваемость»:

Студенты

Номер зачетной книжки	Ф. И. О.	Серия паспорта	Номер паспорта
55500	Иванов Иван Петрович	0402	645327
55800	Климов Андрей Иванович	0402	673211
55865	Новиков Николай Юрьевич	0202	554390

Успеваемость

Номер зачетной книжки	Предмет	Учебный год	Семестр	Оценка
55500	Физика	2013/2014	1	5
55500	Математика	2013/2014	1	4
55800	Физика	2013/2014	1	4
55800	Физика	2013/2014	2	5

Строки таких таблиц называются **записями,** а столбцы – **полями**. На пересечении строк и столбцов должны находиться «атомарные» значения, которые нельзя разбить на какие-либо элементы без потери смысла.

В теории баз данных эти таблицы называют **отношениями (relation)** – поэтому и базы данных называются реляционными. Отношение – это математический термин. При определении свойств таких отношений используется теория множеств. В терминах данной теории строки таблицы будут называться **кортежами**, а колонки – **атрибутами**. Отношение имеет заголовок, который состоит из атрибутов, и тело, состоящее из кортежей. Количество атрибутов называется **степенью отношения**, а количество кортежей – **кардинальным числом.**

При работе с базой данных часто приходится следовать различным ограничениям. В нашем случае ограничения следующие:

– номер зачетной книжки состоит из пяти цифр и не может быть отрицательным;

– номер семестра может принимать только два значения – 1 и 2;

– оценка может принимать только три значения – 3, 4 и 5.

Для идентификации строк в таблицах и для связи таблиц между собой используются так называемые ключи. **Потенциальный ключ** – это уникальный идентификатор строки в таблице базы данных. Он состоит из одного или нескольких полей этой строки. Например, в таблице «Студенты» таким идентификатором может быть поле «Номер зачетной книжки», а могут быть и два поля, взятые вместе, – «Серия паспорта» и «Номер паспорта». В последнем случае ключ будет составным. При этом важным является то, что потенциальный ключ должен быть *не избыточным*, т. е. никакое подмножество полей, входящих в него, не должно обладать свойством уникальности. В нашем примере ни поле «Серия паспорта», ни поле «Номер паспорта» в отдельности не могут использоваться в качестве уникального идентификатора.

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

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

Для обеспечения ограничений ссылочной целостности применяются специальные способы проектирования базы данных. Предусматривается, что при удалении записи из ссылочной таблицы соответствующие записи из ссылающейся таблицы должны быть также удалены, а при изменении значения поля, на которое ссылается внешний ключ, должны быть изменены значения внешнего ключа в ссылающейся таблице. Этот подход называется **каскадным удалением (обновлением).**

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

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

2.2. Определение данных и манипулирование данными

Язык SQL – это непроцедурный язык, который является стандартным средством работы с данными во всех реляционных СУБД. Операторы (команды), написанные на этом языке, лишь указывают СУБД, какой результат должен быть получен, но не описывают процедуру получения этого результата. СУБД сама определяет способ выполнения команды пользователя. В языке SQL традиционно выделяются операторы определения данных (data definition) и операторы манипулирования данными (data manipulation).

К операторам определения данных относятся команды для создания, изменения и удаления таблиц, команды для создания схем в базе данных, а также команды для управления привилегиями доступа к объектам базы данных.

К операторам манипулирования данными относятся команды для вставки записей в таблицы, обновления и удаления записей.

Предлагаем вам кратко ознакомиться с языком SQL на практике. Для выполнения этой задачи сначала создайте базу данных с помощью команды

```
/usr/local/pgsql/bin/createdb test
```

Имя базы данных вы можете выбрать другое, отличное от test.

Теперь запустите утилиту psql и подключитесь к этой базе данных с учетной записью одного из пользователей СУБД, созданных вами ранее, например, my_user:

```
psql -d test –U my_user
```

Для создания таблиц в языке SQL служит команда CREATE TABLE. Ее упрощенный синтаксис таков:

```
CREATE TABLE имя_таблицы
( имя_поля тип_данных [ограничения_целостности],
  имя_поля тип_данных [ограничения_целостности],
  ...
  имя_поля тип_данных [ограничения],
  [первичный_ключ]
);
```

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

Для получения в среде утилиты psql полной информации о команде языка SQL введите команду

```
\h CREATE TABLE
```

Создайте таблицу «Студенты», описанную в начале этой главы. Таблица имеет следующую структуру (т. е. набор полей и их типы данных):

Имя поля	Тип данных	Тип данных PostgreSQL
Номер зачетной книжки	Числовой	numeric(5)
Ф. И. О.	Символьный	text
Серия паспорта	Числовой	numeric(4)
Номер паспорта	Числовой	numeric(6)

Число в описании типа данных numeric означает количество цифр, которые можно ввести в это поле. Тип данных text позволяет ввести значение, содержащее любые символы. Для поля «Серия паспорта» мы выбрали числовой тип, хотя более дальновидным был бы выбор символьного типа (см. задание 3 в конце главы).

```
CREATE TABLE students
( mark_book numeric( 5 ) NOT NULL,
  name text NOT NULL,
  psp_ser numeric( 4 ),
  psp_num numeric( 6 ),
  PRIMARY KEY ( mark_book )
);
```

Для СУБД регистр символов (прописные или строчные буквы) значения не имеет. Однако традиционно ключевые слова языка SQL вводят в верхнем регистре, что повышает наглядность SQL-операторов.

Примечание. Эту команду (как и все SQL-команды в утилите psql) можно вводить двумя способами. Первый способ заключается в том, что команда вводится полностью на одной строке, при этом строка сворачивается «змейкой». Нажимать клавишу Enter после ввода каждого фрагмента команды не нужно, но можно для повышения наглядности вводить пробел. Этот способ удобнее тем, что он позволяет впоследствии с помощью клавиши «стрелка вверх» вызвать на экран (из буфера истории введенных команд) всю команду полностью и при необходимости отредактировать ее либо выполнить еще раз без редактирования. Мы рекомендуем вам использовать именно этот способ.

Второй способ заключается в построчном вводе команды точно так же, как она напечатана в тексте главы. При этом после ввода каждой строки нужно нажимать клавишу Enter. До тех пор пока команда не введена полностью, вид приглашения к вводу команд, выводимого утилитой psql, будет отличаться от первоначального. В конце команды необходимо поставить точку с запятой. Недостаток этого способа в том, что впоследствии вызвать из истории команд всю эту команду целиком невозможно, придется вызывать ее построчно, что не очень удобно.

В приведенной команде выражение NOT NULL означает, что не допускается ввод таких записей в таблицу students, у которых значение поля mark_book или name не указано. Выражение PRIMARY KEY (mark_book) означает, что поле mark_book будет служить первичным ключом таблицы students, т. е. значения этого поля будут уникальными идентификаторами записей в таблице.

Если вы ввели эту команду правильно, то система выдаст сообщение об успешном создании таблицы:

```
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index
"students_pkey" for table "students"
CREATE TABLE
```

Это сообщение говорит также и о том, что СУБД создала индекс для поддержки этого первичного ключа.

Теперь введите несколько записей в созданную таблицу. Упрощенный формат команды таков:

```
INSERT INTO имя_таблицы ( имя_поля, имя_поля ... )
VALUES ( значение_поля, значение_поля, ... );
```

В нашем случае это будет выглядеть так:

```
INSERT INTO students ( mark_book, name, psp_ser, psp_num )
VALUES ( 12300, 'Иванов Иван Иванович', 0402, 543281 );
```

Обратите внимание на одинарные кавычки, в которые заключено значение поля name. Для символьных полей кавычки обязательны, а для числовых – нет. Введите 2–3 записи, изменяя значения полей. Вспомните о том, что можно редактировать ранее введенную команду, вызвав ее на экран при помощи клавиша «стрелка вверх».

Для выборки информации из таблиц базы данных служит такая команда:

```
SELECT имя_поля, имя_поля ...
FROM имя_таблицы;
```

Выберем всю информацию из таблицы students:

```
SELECT * FROM students;
```

Символ «*» означает выбор ВСЕХ полей каждой записи:

```
test=# SELECT * FROM students;
 mark_book |          name           | psp_ser | psp_num
-----------+-------------------------+---------+---------
     12300 | Иванов Иван Иванович    |     402 |  543281
     12302 | Сидоров Сидор Сидорович |     402 |  543381
     12301 | Петров Петр Петрович    |     202 |  543285
(3 rows)
```

Можно выбрать только фамилии, имена и отчества студентов:

```
SELECT name FROM students;
```

```
test=# SELECT name FROM students;
          name
------------------------
 Иванов Иван Иванович
```

```
 Сидоров Сидор Сидорович
 Петров Петр Петрович
(3 rows)
```

В том случае, когда вы вводите значения ВСЕХ полей, можно не перечислять их имена в первой части команды INSERT. Однако в этом случае нужно строго соблюдать порядок полей, т. е. тот порядок, в котором они были описаны при создании таблицы.

Введите еще 2–3 записи, изменяя значения полей, используя сокращенную форму команды INSERT:

```
INSERT INTO students
VALUES ( 12700, 'Климов Андрей Николаевич', 0204, 123281 );
```

Теперь введите еще 2–3 записи, изменяя значения полей, используя сокращенную форму команды INSERT, при которой вводятся значения НЕ ВСЕХ полей:

```
INSERT INTO students ( mark_book, name )
VALUES ( 13200, 'Павлов Павел Павлович' );
```

В этой команде используются лишь те поля, для которых указано ограничение NOT NULL в команде создания таблицы. Поля, для которых такое ограничение указано, должны ОБЯЗАТЕЛЬНО присутствовать в команде INSERT.

Попробуйте ввести такую запись:

```
INSERT INTO students ( mark_book, psp_ser, psp_num )
VALUES ( 13700, 0402, 432781 );
```

Вы получите сообщение об ошибке (внимательно изучите его):

```
ERROR:  null value in column "name" violates not-null con-
straint
```

Попробуйте ввести запись с таким значением поля «Номер зачетной книжки» (mark_book), которое вы уже вводили. Вы также получите сообщение об ошибке. Подумайте, почему оно появилось.

Для того чтобы выбрать из таблицы не все записи, а только те, которые удовлетворяют какому-либо условию, используется более сложная форма команды SELECT:

```
SELECT имя_поля, имя_поля ...FROM имя_таблицы WHERE условие;
```

Например, чтобы выбрать запись для студента с номером зачетной книжки 12300, выполните команду:

```
SELECT * FROM students WHERE mark_book = 12300;
```

Если вы помните, то в таблице «Студенты» (students) есть несколько записей, у которых заполнены значения не всех полей. Для того чтобы эти значения заполнить, сначала нужно выявить такие записи. Для этого можно поступить следующим образом:

```
SELECT * FROM students ORDER BY name;
```

Предложение ORDER BY служит для сортировки записей. С помощью данной команды вы не только выберете записи из таблицы, но также упорядочите их по значениям поля «Ф. И. О.» (name). Вы увидите все записи из таблицы students и среди них – искомые записи с пустыми значениями полей psp_ser и psp_num. Однако в том случае, когда записей в таблице много, такой способ не очень удобен. Лучше поступить так:

```
SELECT * FROM students WHERE psp_ser IS NULL;
```

Обратите внимание на выражение: psp_ser IS NULL. Оно принципиально отличается от выражения: psp_ser = '' (здесь введены две одинарные кавычки). Вспомните, что NULL (null) означает отсутствие какого-либо значения вообще, в том числе и пустой строки.

Теперь обновите записи, в которых не указаны паспортные данные. Для этого воспользуйтесь командой:

```
UPDATE имя_таблицы
SET имя_поля=значение, имя_поля=значение, ...
WHERE условие;
```

Условие, указываемое в команде, должно ограничить диапазон обновляемых записей. Для того чтобы ввести паспортные данные для студента с номером зачетной книжки, например, 12300, нужно выполнить такую команду:

```
UPDATE students
SET psp_ser=0402, psp_num=123456
WHERE mark_book=12300;
```

Команда удаления записей похожа на команду выборки:

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

Удалите какую-нибудь одну запись из таблицы «Студенты» (students):

```
DELETE FROM students WHERE mark_book = 12300;
```

Вы можете указать и какое-нибудь более сложное условие, например:

```
DELETE FROM students
WHERE mark_book >= 12300 AND mark_book <= 12500;
```

или

```
DELETE FROM students
WHERE name <> 'Иванов Иван Иванович' AND mark_book <= 12500;
```

В этой команде символы «<>» означают, что значения поля name в удаляемых записях должны быть не равны указанному значению.

Примечание. Если вы удалили слишком много записей, восстановите их, используя возможность редактировать ранее введенные команды (клавиш «стрелка вверх»).

Теперь создайте ту таблицу «Успеваемость», которая была описана в начале главы. Структура ее такова:

Имя поля	Тип данных	Тип данных PostgreSQL
Номер зачетной книжки	Числовой	numeric(5)
Предмет	Символьный	text
Учебный год	Символьный	text
Семестр	Числовой	numeric(1)
Оценка	Числовой	numeric(1)

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

Команда для создания этой таблицы более сложная, чем предыдущая. Дополнительные ограничения CHECK позволяют ограничить допустимые значения полей term и mark. Предложение DEFAULT позволяет указать значение по умолчанию для поля mark. Предложение FOREIGN KEY создает ограничение ссылочной целостности и указывает в качестве ссылочного ключа поле mark_book. Это означает, что в таблицу «Успеваемость» (progress) нельзя ввести запись, значение поля mark_book которой отсутствует в таблице «Студенты» (students). Говоря простым языком, нельзя ввести запись об оценке того студента, информация о котором еще не введена в таблицу «Студенты». При удалении записи из таблицы «Студенты» (students) будут также удалены записи из таблицы «Успеваемость» (progress), у которых значение поля mark_book совпадает со значением этого поля в удаляемой записи в таблице «Студенты». Таким образом, при удалении информации о студенте удаляется и информация обо всех его оценках. Если же мы изменим номер зачетной книжки в таблице «Студенты», то СУБД сама изменит этот номер во всех записях об оценках для данного студента. Подобные действия, которые выполняет сама СУБД, называются каскадным удалением и каскадным обновлением соответственно.

Для того чтобы посмотреть, какая получилась таблица, введите команду

```
\d progress
```

Это не команда языка SQL, а команда утилиты psql. Она служит для вывода информации о структурах таблиц.

Введите несколько записей в таблицу «Успеваемость» с помощью команды INSERT. Можно воспользоваться такой формой команды, в которой не указываются имена полей, но в этом случае нужно задать значения всех полей в том порядке, в котором они перечислены в команде создания таблицы CREATE TABLE.

```
INSERT INTO progress VALUES (12500, 'Физика', '2013/2014', 1, 4);
```

Введите несколько записей (по 2–3 для каждого из студентов, перечисленных в таблице «Студенты»), изменяя, соответственно, номер зачетной книжки, предмет, семестр и оценку.

Попробуйте ввести предшествующую команду, сократив ее немного:

```
INSERT INTO progress VALUES (12700,'Математика','2013/2014', 1);
```

Отличие в том, что для поля «Оценка» (mark) значение не указывается. Это допустимо, так как данное поле идет в списке последним. А поскольку для него предусмотрено значение по умолчанию (5), то именно это значение и будет записываться в базу данных, если вы не укажете значение явно. Если бы такое поле (для которого задано значение по умолчанию) было не последним в списке, то нам пришлось бы тогда использовать полную форму команды INSERT:

```
INSERT INTO имя_таблицы ( имя_поля, имя_поля ... )
VALUES ( значение_поля, значение_поля, ... );
```

Но в списке имен полей и в списке значений полей мы могли бы пропустить то поле, которое имеет значение по умолчанию.

Попробуйте ввести в таблицу «Успеваемость» (progress) запись, у которой значение поля «Номер зачетной книжки» (mark_book) такое, которого нет в таблице «Студенты» (students). Вы получите сообщение об ошибке, которое демонстрирует работу правил ссылочной целостности:

```
ERROR:  insert or update on table "progress" violates foreign
key constraint "progress_mark_book_fkey"
DETAIL:  Key (mark_book)=(99900) is not present in table "stu-
dents".
```

Теперь решим более сложную задачу. Предположим, что нам нужно получить экзаменационную ведомость по физике (или другому предмету) за первый семестр 2013/2014 учебного года. Для этого мы выполняем такую команду:

```
SELECT * FROM students, progress
WHERE progress.acad_year = '2013/2014' AND
      progress.term = 1 AND
      progress.subject = 'Физика' AND
      students.mark_book = progress.mark_book;
```

Обратите внимание на наличие имен двух таблиц в предложении FROM, а также на строку students.mark_book = progress.mark_book в предложении WHERE. Данная строка (условие) позволяет выполнить так называемое **соединение** таблиц. При этом формируются объединенные записи из тех записей двух таблиц, у которых значения поля mark_book одинаковы.

В экзаменационных ведомостях не требуется наличия паспортных данных. В них можно также не указывать для каждого студента

повторяющееся наименование предмета, поэтому мы можем указать в команде только часть полей. Отсортируем нашу ведомость по фамилии, имени и отчеству студентов.

```
SELECT progress.mark_book, students.name, progress.mark
FROM students, progress
WHERE progress.acad_year = '2013/2014' AND
      progress.term = 1 AND
      progress.subject = 'Физика' AND
      students.mark_book = progress.mark_book
ORDER BY students.name;
```

А теперь мы можем выбрать все оценки для одного студента за весь период обучения:

```
SELECT *
FROM progress
WHERE mark_book = ( SELECT mark_book
                    FROM students
                    WHERE name = 'Иванов Иван Иванович' )
ORDER BY acad_year, term, subject;
```

Обратите внимание, что в приведенной команде не указываются имена таблиц перед именами полей. Это допустимо, когда СУБД сможет однозначно определить, из какой таблицы выбирать конкретное поле. Обратите также внимание на наличие двух предложений SELECT в одной команде. Второе предложение SELECT называется **подзапросом**. Этот подзапрос дает значение поля mark_book для того студента, значение поля «Ф. И. О.» (name) которого равно 'Иванов Иван Иванович'.

Для проверки работы правил ссылочной целостности выполните обновление поля mark_book в одной из записей таблицы students:

```
UPDATE students SET mark_book = 12900 WHERE mark_book = 12300;
```

Сделайте выборку записей из таблицы students и убедитесь в том, что обновление произошло. Затем сделайте выборку записей из обеих таблиц, как вы делали это ранее, и убедитесь, что и в таблице progress обновление также произошло. Это было выполнено благодаря работе правил ссылочной целостности. Причем данные операции были произведены в рамках одной **транзакции**. Если бы в процессе их выполнения произошел сбой электропитания, то в базе данных все записи остались бы в том состоянии, в котором они находились до начала выполнения вашей команды UPDATE.

Выполните команду удаления какой-нибудь записи из таблицы students:

```
DELETE FROM students WHERE mark_book = 12300;
```

Сделайте выборку записей из обеих таблиц, как вы делали это ранее, и убедитесь, что и из таблицы progress записи были также удалены.

Можно сделать и общую выборку из таблицы progress и визуально убедиться в том, что удаление записей действительно имело место:

```
SELECT * FROM progress;
```

Для выхода из программы psql наберите команду \q и нажмите Enter.

2.3. Индексы

Индекс – это специальная структура данных в базе данных, которая позволяет ускорить процесс отыскания и извлечения строк из таблиц базы данных. Однако индексы требуют и некоторых накладных расходов на их создание и поддержание в процессе обновлений данных в таблицах. Поэтому использовать индексы нужно осмотрительно.

Для создания индекса используется команда:

```
CREATE INDEX имя_индекса ON имя_таблицы ( имя_поля, …);
```

Например, создадим индекс по столбцу name для таблицы students:

```
CREATE INDEX students_name ON students ( name );
```

Для того чтобы увидеть индексы, созданные для данной таблицы, нужно воспользоваться командой утилиты psql:

```
\d имя_таблицы
```

Просмотреть список всех индексов в текущей базе данных можно с помощью утилиты psql, введя команду

```
\di
```

или

```
\di+
```

Для удаления индекса используется команда:

```
DROP INDEX имя_индекса;
```

Когда индекс уже создан, о его поддержании в актуальном состоянии заботится СУБД. Индекс, созданный по столбцу, участвующему в соединении двух таблиц, позволит ускорить процесс выборки записей из таблиц. При выборке записей в отсортированном порядке индекс также может помочь, если сортировка выполняется по тем полям, по которым индекс создан.

PostgreSQL может создавать индексы различных типов, но по умолчанию команда CREATE INDEX использует так называемое B-дерево. Такой индекс подходит для большинства типовых задач. Здесь мы будем рассматривать только индексы на основе B-дерева.

Индексы могут создаваться не только по одному столбцу, но и по нескольким. Однако такие индексы по двум и более столбцам следует использовать очень экономно (осторожно). Как правило, бывает вполне достаточно индекса по одному столбцу. Такой индекс позволяет сэкономить время на его поддержание и занимаемое им место. Индексы более чем по трем столбцам могут быть полезны только в специальных случаях.

Если в SQL-запросе есть предложение ORDER BY, то индекс может позволить избежать этапа сортировки выбранных строк. Однако если SQL-запрос просматривает значительную часть таблицы, то явная сортировка выбранных строк может оказаться быстрее, чем использование индекса. Индексы более полезны, когда из таблицы выбирается лишь несколько строк. В случае использования предложения ORDER BY в комбинации с LIMIT n явная сортировка (при отсутствии индекса) потребует обработки всех строк таблицы ради того, чтобы определить первые *n* строк. Но если есть индекс по тем же столбцам, по которым производится сортировка ORDER BY, то эти первые *n* строк могут быть извлечены непосредственно, без сканирования остальных строк вообще.

При создании индексов может использоваться возрастающий порядок значений в индексируемом столбце или убывающий. По умолчанию порядок возрастающий, при этом значения NULL идут последними. При создании индекса можно модифицировать поведение

по умолчанию с помощью опций ASC (возрастающий порядок), DESC (убывающий порядок), NULL FIRST и NULLS LAST. Например:

```
CREATE INDEX имя_индекса ON имя_таблицы ( имя_поля NULLS FIRST
);
CREATE INDEX имя_индекса ON имя_таблицы ( имя_поля DESC NULLS
LAST );
```

Индексы могут также использоваться для обеспечения уникальности значений полей в строках таблицы. В таком случае создается уникальный индекс. Для его создания используется команда:

```
CREATE UNIQUE INDEX имя_индекса ON имя_таблицы ( имя_поля, …);
```

Например, создадим уникальный индекс по столбцу name для таблицы students:

```
CREATE UNIQUE INDEX students_name ON students ( name );
```

В этом случае мы уже не сможем ввести в таблицу «Студенты» записи о двух студентах, имеющих одинаковые фамилию, имя и отчество. Конечно, здесь мы не обсуждаем вопрос практической целесообразности наложения такого ограничения на эту таблицу, а говорим лишь о технической возможности создания уникального индекса и последствиях этого шага.

Важно, что в уникальных индексах допускается наличие значений NULL, поскольку они считаются не совпадающими ни с какими другими значениями, в том числе и друг с другом. Если уникальный индекс создан по нескольким полям, то совпадающими считаются лишь те комбинации значений полей двух записей, в которых совпадают значения всех полей.

Если структура таблицы предусматривает создание первичного ключа или содержит ограничение уникальности для каких-то столбцов, тогда СУБД создаст неявные индексы сама и выведет сообщения об этом. Сообщение может быть выведено и на русском языке, если ваша система русифицирована. Например:

```
psql:db.sql:17: ЗАМЕЧАНИЕ:  CREATE TABLE / PRIMARY KEY создаст
неявный индекс "urls_pkey" для таблицы "urls"
psql:db.sql:17: ЗАМЕЧАНИЕ:   CREATE TABLE / UNIQUE создаст
неявный индекс "urls_url_key" для таблицы "urls"
```

Кроме простого указания имен столбцов в команде создания индекса можно использовать также функции и скалярные выражения, построенные на основе столбцов таблицы. Например, если мы хотим запретить значения поля «Модель автомобиля» model, отличающиеся только регистром символов, мы используем команду

```
CREATE UNIQUE INDEX cars_model_names ON cars ( lower( model )
);
```

Встроенная функция lower() преобразует все символы в нижний регистр. Индекс строится уже на основе преобразованных значений. Если мы сначала введем в таблицу значение ‘Toyota’, то значение ‘TOYOTA’ ввести уже будет невозможно.

Индексы на основе выражений требуют больше ресурсов для их создания и поддержания при вставке и обновлении записей в таблице. Зато при выполнении выборок, построенных на основе сложных выражений, работа происходит с меньшими накладными расходами, поскольку в индексе хранятся уже вычисленные значения этих выражений, пусть даже самых сложных. Поэтому такие индексы целесообразно использовать тогда, когда выборки производятся многократно, и время, затраченное на создание и поддержание индекса, компенсируется (окупается) при выполнении выборок из таблицы.

2.4. Управление транзакциями

Механизм реализации транзакций в СУБД PostgreSQL основан на мультиверсионной модели (Multiversion Concurrency Control). Кроме управления транзакциями PostgreSQL позволяет также создавать явные блокировки данных как на уровне отдельных строк таблиц, так и на уровне целых таблиц.

Для организации выполнения параллельных транзакций с использованием утилиты psql будем использовать такую технологию. Запустите утилиту psql на двух терминалах, на каждом из них организуйте транзакцию с помощью команд BEGIN … END. В рамках первой транзакции выполните команду, например, для блокировки конкретной строки таблицы:

```
SELECT * FROM students WHERE mark_book = … FOR UPDATE;
```

На втором терминале также в рамках транзакции выполните эту же команду. Важно, чтобы значения поля mark_book в условии WHERE были одинаковыми. В этом случае происходит обращение

двух транзакций к одной и той же записи. Вы увидите, что выполнение команды SELECT на втором терминале будет задержано до тех пор, пока вы не выполните команду END на первом терминале, завершив первую транзакцию.

Аналогичным образом организуем блокировки на уровне таблиц. Также на первом терминале начните транзакцию командой BEGIN. Затем выполните команду блокировки всей таблицы в самом строгом режиме

```
LOCK students IN ACCESS EXCLUSIVE MODE;
```

На втором терминале выполните совершенно «безобидную» команду

```
SELECT * FROM students WHERE mark_book = … ;
```

Вы увидите, что выполнение команды SELECT на втором терминале будет задержано. Прервите транзакцию на первом терминале командой ROLLBACK. Вы увидите, что на втором терминале команда будет успешно выполнена.

Теперь перейдем к изучению уровней изоляции транзакций. Команды на первом терминале такие:

```
BEGIN;
SELECT * FROM students WHERE mark_book = … FOR UPDATE;
```

Команды на втором терминале:

```
BEGIN;
SELECT * FROM students WHERE mark_book = …;
```

Вы увидите, что в поле name на обоих терминалах будет одно и то же значение.

Команды на первом терминале:

```
UPDATE students SET name = 'XXXXXX' WHERE mark_book = … ;
SELECT * FROM students WHERE mark_book = … ;
```

Команды на втором терминале:

```
SELECT * FROM students WHERE mark_book = …;
```

Вы увидите, что в поле name на первом терминале будет новое значение, а на втором терминале – старое значение, которое было в этой записи на момент начала второй транзакции.

Команда на первом терминале:

```
END;
```

На первом терминале мы завершили транзакцию, зафиксировав изменения в базе данных. Команда на втором терминале:

```
SELECT * FROM students WHERE mark_book = …;
```

Вы увидите, что теперь в поле name на обоих терминалах будет новое значение, хотя завершилась только первая транзакция, в которой это изменение и было произведено, а вторая транзакция еще не завершена. Таким образом, во второй транзакции становятся видны изменения, выполненные в первой транзакции, только когда они зафиксированы командой END (COMMIT).

Завершите транзакцию и на втором терминале:

```
END;
```

В нашем примере мы проиллюстрировали уровень изоляции транзакций Read Committed, который используется по умолчанию в PostgreSQL. Когда транзакция использует этот уровень изоляции, запрос SELECT (без предложения FOR UPDATE/SHARE) видит только данные, зафиксированные *до начала* выполнения этого запроса. Он не видит незафиксированные данные и данные, зафиксированные параллельными транзакциями уже *в процессе* выполнения этого запроса. Запрос SELECT видит моментальный снимок базы данных, сделанный в момент начала выполнения запроса. Однако этот запрос SELECT видит обновления, сделанные другими запросами в рамках его собственной транзакции, хотя эти изменения еще и не зафиксированы в базе данных. Два последовательных запроса SELECT, выполняемые в рамках одной транзакции, могут увидеть различные данные, если другие транзакции успели зафиксировать изменения во время выполнения первого запроса SELECT.

2.5. Повышение производительности

PostgreSQL создает *план выполнения* для каждого запроса. Этот план должен учитывать структуру запроса и свойства данных для достижения хорошей производительности. Планированием занимается

специальный *планировщик*. Для просмотра плана выполнения запроса служит команда EXPLAIN. Досконально разбираться в той информации, которую выдает команда EXPLAIN, довольно сложно – требуется опыт. Мы изложим лишь основные приемы работы с этой командой.

Структура плана запроса представлена в виде дерева, состоящего из узлов (plan nodes). Узлы на нижних уровнях дерева отвечают за просмотр строк таблиц, эти узлы возвращают строки из таблиц. Существуют различные способы просмотра строк: последовательный и на основе индекса. Если конкретный запрос требует выполнения операций агрегирования, соединения таблиц, сортировки, то над узлами выборки строк будут располагаться дополнительные узлы дерева плана. Команда EXPLAIN выводит по одной строке для каждого узла дерева плана, при этом выводятся оценки стоимости выполнения операций на каждом узле, которые делает планировщик. Могут также выводиться дополнительные строки для конкретных узлов. Самая первая строка плана содержит общую оценку стоимости выполнения данного запроса.

Запустите утилиту psql и введите простой запрос:

```
EXPLAIN SELECT * FROM students;
```

В ответ получим план выполнения запроса:

```
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on students  (cost=0.00..7.20 rows=320 width=69)
(1 строка)
```

Поскольку в этом запросе нет предложения WHERE, он должен просмотреть все строки таблицы, поэтому планировщик выбирает простой последовательный просмотр (Sequential Scan). В скобках приведены важные параметры плана.

Первое число означает оценку времени, которое требуется для того, чтобы приступить к выводу данных, например, время, затрачиваемое на сортировку данных. В нашем примере это время равно нулю.

Второе число – это оценка общей стоимости выполнения запроса. Формируя эту оценку, планировщик исходит из предположения, что данный узел плана запроса выполняется до конца, т. е. извлекаются все имеющиеся строки таблицы. Однако в ряде случаев на практике это может оказаться и не так, если узел-родитель прекратит свою работу досрочно, например, в случае использования в запросе

SELECT предложения LIMIT, которое ограничивает выборку записей из таблицы конкретным их числом. Обе оценки стоимости выполнения выражаются в неких *условных единицах*, которые вычисляются на основе ряда параметров сервера базы данных.

Далее в выводе идет общее число строк, которые должны быть извлечены на данном узле плана (опять же при условии выполнения этого узла до полного завершения). В нашем примере число строк равно 320.

Последним параметром узла плана идет оценка средней ширины строк данных, которые выводятся на данном узле плана запроса. В нашем примере ширина строки данных оценивается в 69 байтов.

Теперь усложним наш запрос, добавив в него сортировку данных:

```
EXPLAIN SELECT * FROM students ORDER BY mark_book;
```

Поскольку выводимые строки плана в утилите psql могут быть очень длинными, мы будем сворачивать их при переносе в текст пособия. Получим такой план:

```
                                 QUERY PLAN
-------------------------------------------------------------------
 Index Scan using students_pkey on students  (cost=0.00..20.05
rows=320 width=69)
(1 строка)
```

В этом плане мы видим, что планировщик предписывает использовать просмотр таблиц с использованием индекса students_pkey. Этот индекс был создан неявно самой СУБД, поскольку при создании таблицы «Студенты» мы предусмотрели первичный ключ. Будут просмотрены все 320 записей таблицы «Студенты». Общая оценка стоимости выполнения данного запроса стала больше, чем у предыдущего запроса, поскольку для упорядочивания выбранных строк привлекается индекс, требуя дополнительной работы СУБД.

Сформируем запрос с предложением WHERE, которое предписывает выбрать всего одну запись из таблицы:

```
EXPLAIN SELECT * FROM students WHERE mark_book = 10000;
```

Получим план, в котором предлагается последовательный перебор записей с дополнительным фильтром, хотя у нас есть индекс по полю mark_book. Это может объясняться тем, что таблица имеет

маленький размер, а также тем, что планировщик еще не собрал достаточно статистики о фактически выполненных запросах к таблице «Студенты».

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on students  (cost=0.00..8.00 rows=1 width=69)
   Filter: (mark_book = 10000::numeric)
(2 строки)
```

А если мы модифицируем запрос с тем, чтобы выбирались, возможно, более одной записи, то получим другую картину.

```
EXPLAIN SELECT * FROM students WHERE mark_book <= 10000;
```

Теперь используется индекс. А оцениваемое число выбираемых записей равно 1.

```
                                  QUERY PLAN
-------------------------------------------------------------------
 Index Scan using students_pkey on students  (cost=0.00..4.27
rows=1 width=69)
   Index Cond: (mark_book <= 10000::numeric)
(2 строки)
```

Усложним условие WHERE, задав диапазон значений поля mark_book от 10000 до 10100:

```
EXPLAIN SELECT * FROM students WHERE mark_book >= 10000 AND
mark_book <= 10100;
```

Теперь планировщик отказывается от использования индекса и предпочитает простой последовательный поиск с дополнительным фильтром по полю mark_book.

```
                                  QUERY PLAN
-------------------------------------------------------------------
----
 Seq Scan on students  (cost=0.00..8.80 rows=101 width=69)
   Filter: ((mark_book >= 10000::numeric) AND (mark_book <=
10100::numeric))
(2 строки)
```

Напишем более сложный запрос – соединение двух таблиц. В таблице «Студенты» находится 320 записей, а в таблице «Успеваемость» – 3840.

```
EXPLAIN SELECT * FROM students s, progress p WHERE s.mark_book
= p.mark_book;
```

В плане выполнения запроса мы видим последовательные просмотры сначала таблицы «Студенты», а затем таблицы «Успеваемость», а в узле самого верхнего уровня выполняется непосредственно само соединение таблиц по условию равенства значений поля mark_book в обеих таблицах.

```
                                QUERY PLAN
-----------------------------------------------------------------
----
 Hash Join  (cost=11.20..136.40 rows=3840 width=110)
   Hash Cond: (p.mark_book = s.mark_book)
   ->  Seq Scan on progress p  (cost=0.00..72.40 rows=3840
width=41)
   ->  Hash  (cost=7.20..7.20 rows=320 width=69)
         ->  Seq Scan on students s  (cost=0.00..7.20 rows=320
width=69)
(5 строк)
```

Ограничим количество выбираемых записей числом 10.

```
EXPLAIN SELECT * FROM students ORDER BY mark_book LIMIT 10;
```

В плане мы видим использование индекса.

```
                                QUERY PLAN
-----------------------------------------------------------------
----
 Limit  (cost=0.00..0.63 rows=10 width=69)
   ->  Index Scan using students_pkey on students
cost=0.00..20.05 rows=320 width=69)
(2 строки)
```

Теперь отсортируем выбранные записи по значению поля name, по которому индекс не создан:

```
EXPLAIN SELECT * FROM students ORDER BY name LIMIT 10;
```

В плане предлагается последовательный просмотр записей, а затем предусматривается сортировка и в конце – ограничение количества выбираемых записей числом 10.

```
                              QUERY PLAN
-----------------------------------------------------------------
 Limit  (cost=14.12..14.14 rows=10 width=69)
```

```
   ->  Sort  (cost=14.12..14.92 rows=320 width=69)
         Sort Key: name
         ->  Seq Scan on students  (cost=0.00..7.20 rows=320
width=69)
(4 строки)
```

В команде EXPLAIN можно указать опцию ANALYZE, что позволит выполнить запрос и вывести на экран фактические затраты времени на выполнение запроса и число фактически выбранных записей.

```
EXPLAIN ANALYZE SELECT * FROM students ORDER BY name LIMIT 10;

                              QUERY PLAN
-----------------------------------------------------------------
Limit  (cost=14.12..14.14 rows=10 width=69) (actual
time=3.828..3.840 rows=10 loops=1)
   ->  Sort  (cost=14.12..14.92 rows=320 width=69) (actual
time=3.825..3.829 rows=10 loops=1)
         Sort Key: name
         Sort Method: top-N heapsort  Memory: 18kB
         ->  Seq Scan on students  (cost=0.00..7.20 rows=320
width=69) (actual time=0.069..0.700 rows=320 loops=1)
 Total runtime: 3.935 ms
(6 строк)
```

Контрольные вопросы и задания

1. Что такое транзакция?
2. Что такое план выполнения запроса и какова его структура?
3. На какой модели основан механизм реализации транзакций с СУБД PostgreSQL?
4. К командам определения данных относится команда ALTER TABLE. Она позволяет модифицировать структуру таблицы базы данных, например, добавить или удалить столбец (поле), изменить тип данных существующего столбца (поля), добавить или удалить ограничение, привязанное как к отдельному столбцу (полю), так и ко всей таблице в целом. Самостоятельно ознакомьтесь с командой ALTER TABLE по документации на PostgreSQL. Попробуйте применить полученные знания к таблицам «Студенты» и «Успеваемость». Для просмотра выполненных модификаций используйте интерактивный терминал psql. Команда:

```
\d имя_таблицы
```

5. Для удаления таблицы из базы данных служит команда DROP TABLE. Самостоятельно ознакомьтесь с этой командой по документации на PostgreSQL.

6. При создании таблиц «Студенты» и «Успеваемость» мы использовали числовой тип данных numeric, при этом мы давали этому типу данных только один параметр – точность (precision). Однако этот тип данных позволяет указывать еще и второй параметр – масштаб (scale), т. е. число цифр после запятой. Тип данных numeric является *точным* типом данных. Кроме того в СУБД PostgreSQL имеются и другие числовые типы. Среди них есть целочисленные типы smallint, integer и bigint, занимающие 2, 4 и 8 байтов соответственно, а также числа с плавающей запятой real и double precision, занимающие 4 и 8 байтов соответственно. Самостоятельно ознакомьтесь с перечисленными типами данных по документации на PostgreSQL. Предложите дополнительные поля в таблицы «Студенты» и «Успеваемость», которые могли бы использовать указанные типы данных, либо предложите замену типа numeric в этих таблицах на целочисленные типы.

7. Существуют типы данных для хранения даты и времени. Основной из них – это тип date – только дата. Тип timestamp позволяет хранить в одном поле дату и время. Самостоятельно ознакомьтесь с перечисленными типами данных по документации на PostgreSQL. Обратите внимание на разнообразие допустимых форматов даты и времени. Например, рекомендуемый формат для ввода даты в командах INSERT и UPDATE такой: ‘2014-05-24’, т. е. сначала идет год, затем месяц, а последним – день. Добавьте в таблицу «Успеваемость» столбец (поле) «Дата сдачи зачета/экзамена» с помощью команды ALTER TABLE. Затем введите в эту таблицу несколько новых записей, указывая также и дату сдачи зачета или экзамена.

Для просмотра выполненных модификаций таблицы используйте интерактивный терминал psql. Команда:

```
\d имя_таблицы
```

8. При создании таблиц «Студенты» и «Успеваемость» мы использовали символьный тип text. Однако имеются еще два символьных типа: varchar(n) и char(n). Здесь *n* означает число символов, которые могут быть сохранены в символьной строке, сохраняемой в поле такого типа. Обратите внимание, что речь идет именно о числе символов, а не о числе байтов. Это различие важно при использовании многобайтовой кодировки символов, например, UTF-8.

Подумайте над возможной заменой числового типа данных, выбранного нами для поля «Серия паспорта» в таблице «Студенты», на символьный тип. Обратите внимание на то, что при вводе, например, серии «0402» первый ноль не сохраняется в базе данных.

9. Модифицируйте нашу простую базу данных, включающую только две таблицы: «Студенты» и «Успеваемость». Создайте еще одну таблицу – «Учебные дисциплины». Эта таблица должна содержать два поля: «Код учебной дисциплины» (числовой тип данных) и «Наименование учебной дисциплины» (символьный тип данных). В таблице «Успеваемость» замените поле «Предмет» полем «Код учебной дисциплины». При выводе информации из таблицы «Успеваемость» используйте соединение этой таблицы с таблицей «Учебные дисциплины» в команде SELECT, чтобы вместо числовых кодов на экран выводились наименования учебных дисциплин (предметов).

10. Подумайте над тем, не нужно ли создать в таблице «Успеваемость» первичный ключ. Какие поля (столбцы) вы в него включили бы и почему? Если это целесообразно сделать, то создайте новую версию этой таблицы и введите несколько записей. При вводе попробуйте нарушить требование первичного ключа. Какое сообщение выдаст СУБД?

11. Язык SQL предлагает программисту множество различных функций и операторов: математических, логических, строковых, агрегирующих, функций для обработки и форматирования дат и т. д. Самостоятельно ознакомьтесь с этой темой по документации на PostgreSQL. Обратите внимание на так называемые оконные функции (window functions). Напишите несколько SQL-запросов к таблицам «Студенты» и «Успеваемость» с использованием изученных вами функций. Добавьте в таблицу «Студенты» столбец «Студенческая группа». Напишите SQL-запрос, позволяющий вывести средний балл, полученный студентами каждой группы за конкретный семестр. Подумайте, каким образом можно в данной предметной области (успеваемость студентов) использовать оконные функции. Предложите конкретный пример.

12. В таблицах базы данных могут быть и значения NULL, т. е. отсутствие конкретного значения. Этот случай также может быть учтен при создании индексов. Самостоятельно ознакомьтесь с этой темой по документации на PostgreSQL. Создайте таблицу, в которой будут допустимы значения NULL. Создайте для нее индекс с учетом этого факта. Сделайте выборку данных из таблицы, выведя строки со значениями NULL первыми или последними.

13. Создайте индекс по двум столбцам, причем по одному из них укажите убывающий порядок значений столбца, а по другому – возрастающий. Значения NULL у первого столбца должны располагаться в начале, а у второго – в конце. Просмотрите полученные индексы с помощью команд утилиты psql

```
\d имя_таблицы
\di+ имя_индекса
```

Обратите внимание, что первая команда выведет не только имя индекса, но также и имена столбцов, по которым он создан, а вторая команда выведет размер индекса.

14. В сложных базах данных целесообразно использование комбинаций индексов. Иногда бывают более полезны комбинированные индексы по нескольким столбцам, чем отдельные индексы по единичным столбцам. В реальных ситуациях часто приходится делать выбор, т. е. находить компромисс, между, например, созданием двух индексов по каждому из двух столбцов таблицы либо созданием одного индекса по двум столбцам этой таблицы, либо созданием всех трех индексов. Выбор зависит от того, запросы какого вида будут выполняться чаще всего. Предложите какую-нибудь таблицу базы данных и смоделируйте ситуации, в которых вы приняли бы одно из этих трех возможных решений. Воспользуйтесь документацией на PostgreSQL.

15. Предложите какую-нибудь таблицу и смоделируйте ситуацию, в которой было бы целесообразно использование индекса на основе функции или скалярного выражения от двух или более столбцов.

16. Существует такая возможность, как создание частичных индексов (partial indexes). Самостоятельно ознакомьтесь с этой темой по документации на PostgreSQL. Предложите какую-нибудь ситуацию, в которой частичный индекс был бы полезен. Создайте такую таблицу и такой индекс.

17. Самостоятельно детально ознакомьтесь с управлением транзакциями по документации на PostgreSQL, раздел «Concurrency Control». Изучите все остальные режимы блокировки на уровне отдельных строк и целых таблиц. Проделайте необходимые команды на двух терминалах psql аналогично той технологии, которая показана в пособии.

18. Изучите остальные уровни изоляции транзакций и проиллюстрируйте их путем выполнения необходимых команд в рамках

параллельных транзакций на двух терминалах psql. Попробуйте организовать три и более параллельных транзакции с использованием терминала psql.

19. Команда EXPLAIN имеет опцию BUFFERS. Ознакомьтесь с ней самостоятельно по документации на PostgreSQL, раздел «Performance Tips».

20. Выполните с помощью команды EXPLAIN более сложные запросы, чем те, которые мы рассмотрели в тексте этой главы пособия. Постарайтесь понять, что предлагает планировщик на каждом узле дерева плана. Попробуйте создавать индексы для тех полей, по которым производится сортировка, а затем с помощью команды EXPLAIN ANALYZE посмотрите, насколько быстрее стал выполняться запрос.

21. Планировщик в своей работе использует системные таблицы pg_class и pg_stats. Ознакомьтесь с ними самостоятельно по документации на PostgreSQL, раздел «Performance Tips».

22. При массовом вводе данных в базу данных производительность СУБД может снижаться по ряду причин, например, при наличии индексов они обновляются при вводе каждой новой записи в таблицу, а это требует дополнительных «усилий» со стороны СУБД. Для повышения производительности СУБД в подобных ситуациях в документации предлагается ряд мер, например, удаление индексов перед началом массового ввода данных и пересоздание индексов после завершения такого ввода. Ознакомьтесь с этими мерами самостоятельно по документации на PostgreSQL, раздел «Performance Tips». Смоделируйте ситуации, описанные в этом разделе документации, и выполните рекомендуемые действия.

3. АДМИНИСТРИРОВАНИЕ СЕРВЕРА БАЗ ДАННЫХ

3.1. Конфигурирование сервера баз данных

Конфигурирование сервера PostgreSQL осуществляется за счет изменения значений множества параметров. Все имена параметров являются нечувствительными к регистру символов, поэтому их можно вводить как в верхнем, так и в нижнем регистрах. Параметры могут принимать значения пяти типов: булевский, целый, с плавающей точкой, строковый и перечислимый. При этом параметры булевского типа могут принимать значения в различных формах: on, off, true, false, yes, no, 1, 0. Допускаются также и t, f, y, n.

Ряд параметров касаются объема памяти или времени. Они имеют единицы измерения, используемые по умолчанию, например, kB (килобайты), ms (миллисекунды). При необходимости, задавая значения параметров, можно указывать явно и другие единицы измерения, например, MB (мегабайты), s (секунды) и т. д. Единицы измерения, используемые по умолчанию, можно найти в системной таблице pg_settings (поле units) с помощью команды:

```
SELECT name, setting, unit FROM pg_settings;
```

Параметры перечислимого типа аналогичны строковым параметрам, но могут принимать лишь ограниченный набор значений, который также можно найти в системной таблице pg_settings (поле enumvals).

Назначить параметрам значения можно несколькими способами. Самый первый из них – это использование конфигурационного файла postgresql.conf, который находится в том же каталоге, что и файлы базы данных. Если вы установили PostgreSQL в каталоги по умолчанию, тогда файл postgresql.conf будет находиться в каталоге /usr/local/pgsql/data. Формат файла таков:

```
# This is a comment
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

Для того чтобы изменения, произведенные в этом файле, вступили в силу, необходимо перезагрузить сервер с помощью команды

```
/usr/local/pgsql/bin/pg_ctl  reload
```

Этого же эффекта можно добиться путем отправки серверному процессу сигнала SIGHUP напрямую (команда вводится на одной строке):

```
kill -1 `cat /usr/local/pgsql/data/postmaster.pid |
 sed -n -e '1p'`
```

Существует целый ряд параметров, для вступления которых в силу потребуется полный перезапуск сервера баз данных. Такие параметры помечены в файле postgresql.conf примечанием «change requires restart».

Второй способ задания параметров сервера, это использование параметров в командной строке при запуске сервера. Например:

```
postgres -c log_connections=yes -c log_destination='syslog'
```

Изменения параметров, выполняемые таким образом, имеют превосходство над значениями параметров, установленными в конфигурационном файле postgresql.conf.

Третий способ – назначить новые значения параметров конкретной учетной записи пользователя или базе данных. Это выполняется с помощью команд ALTER ROLE или ALTER DATABASE. Установки, сделанные для конкретной базы данных, имеют преимущество над теми, которые были назначены в конфигурационном файле postgresql.conf или в командной строке. Речь идет, конечно, о ситуации, когда *разные значения* назначены *одним и тем же параметрам*. Установки, сделанные для конкретной учетной записи пользователя, имеют преимущество над теми, которые были сделаны для конкретной базы данных.

Еще один способ, который, правда, допустим не для всех параметров, – назначить значение параметра только для конкретного сеанса работы с клиентской утилитой. Для этого служит переменная среды PGOPTIONS. Например, чтобы изменить формат вывода дат в утилите psql, нужно сделать так:

```
PGOPTIONS='-c datestyle=SQL' psql -d test -U postgres
```

Данный способ также имеет преимущество над всеми предшествующими в случае конфликтующих значений параметров.

И последний способ изменений значений параметров – это команда SET, например, для изменения формата дат в утилите psql можно выполнить команду:

```
SET datestyle TO postgres, dmy;
```

Этот способ имеет наивысший приоритет, но такие установки действуют только в течение текущего сеанса работы с программой.

Просмотреть значения параметров можно с помощью команды SHOW. Она позволяет вывести значение конкретного параметра или всех параметров, например:

```
SHOW datestyle;
SHOW ALL;
```

Однако самая полная информация о параметрах системы содержится в виртуальной таблице pg_settings.

Получить информацию о системе можно также с помощью множества специальных функций, представленных в главах «System Information Functions» и «System Administration Functions» документации. Представителями таких функций являются current_user (имя текущего пользователя) и version() (версия СУБД):

```
SELECT current_user;
SELECT version();
```

Обратите внимание, что у первой функции отсутствуют круглые скобки.

И в завершение укажем на еще один источник информации о системе. Это так называема информационная схема, содержащая несколько десятков представлений (view). Она автоматически создается в каждой базе данных. Для обращения к представлениям, содержащимся в ней, нужно в команде SELECT указывать имя этой схемы. Например, для всех таблиц текущей базы данных можно получить информацию об именах столбцов и их типах данных:

```
SELECT table_name, column_name, data_type
 FROM information_schema.columns;
```

3.2. Аутентификация клиентов

Аутентификация – это процедура, с помощью которой сервер баз данных устанавливает подлинность клиента и определяет, разрешено ли клиенту с таким именем устанавливать соединение с сервером. PostgreSQL предлагает несколько методов аутентификации клиентов. Конкретный метод может быть выбран на основе адреса

компьютера пользователя, его имени и конкретной базы данных. Имена пользователей СУБД PostgreSQL логически отделены от имен пользователей операционной системы, в среде которой функционирует сервер. Поэтому эти имена вовсе не обязаны совпадать. Например, в операционной системе обязательно есть пользователь с именем root, а в PostgreSQL такой пользователь быть не обязан. Однако при совпадении этих имен при установлении соединения с сервером можно не указывать имя пользователя базы данных.

Аутентификация клиентов осуществляется с помощью специального файла pg_hba.conf, который находится в том же каталоге, что и файлы базы данных. Если вы установили PostgreSQL в каталоги по умолчанию, тогда файл pg_hba.conf будет находиться в каталоге /usr/local/pgsql/data. Записи в этом файле могут иметь один из семи форматов. Например, следующая строка разрешает всем локальным пользователям подключаться к любой базе данных на сервере:

```
# TYPE  DATABASE        USER            ADDRESS
METHOD
local   all             all
trust
```

Если мы хотим, чтобы сервер запрашивал пароль при попытке локального пользователя подключиться к нему, тогда в последней колонке мы должны указать ключевое слово «password»:

```
# TYPE  DATABASE        USER            ADDRESS
METHOD
local   all             all
password
```

Конечно, это самый простой метод ограничения доступа к серверу баз данных. Пароль передается в незашифрованном виде. Если предполагается подключение пользователей не с локального компьютера, а по сети, тогда метод «password» использовать нельзя, нужно использовать метод «md5». Для того чтобы изменения, внесенные в файл pg_hba.conf, вступили в действие, нужно поступить так же, как мы делали ранее при внесении изменений в файл postgresql.conf: послать серверному процессу сигнал SIGHUP или воспользоваться командой

```
/usr/local/pgsql/bin/pg_ctl  reload
```

3.3. Управление учетными записями и привилегиями пользователей

PostgreSQL управляет разграничением прав доступа к базам данных, используя концепцию *ролей*. Роль может представлять собой отдельного пользователя базы данных либо группу пользователей, в зависимости от того, как эта роль была создана. Важно то, что роли являются глобальными для всего кластера баз данных, работающего под управлением сервера баз данных. Это значит, что созданная роль может использоваться со всеми базами данных этого сервера. Для создания новой роли служит команда

```
CREATE ROLE имя_роли;
```

Удалить роль можно командой

```
DROP ROLE имя_роли;
```

Можно использовать и аналогичные утилиты PostgreSQL:

```
/usr/local/pgsql/bin/createuser имя_пользователя
/usr/local/pgsql/bin/dropuser имя_пользователя
```

Посмотреть перечень ролей, созданных на сервере, можно двумя способами. Первый из них – команда утилиты psql:

```
\du
```

Второй способ – обращение к системной таблице pg_roles:

```
SELECT rolname FROM pg_roles;
```

Конкретные роли могут владеть объектами базы данных и предоставлять права (привилегии) доступа к этим объектам другим ролям. Обычно владельцем конкретного объекта, например, таблицы, является та роль, которая создала этот объект. Перечень видов привилегий доступа довольно разнообразен. С таблицами базы данных наиболее часто используются следующие права доступа: право на выполнение запросов – SELECT, право на выполнение вставки записей – INSERT, право на удаление записей – DELETE, право на обновление записей – UPDATE. Для предоставления прав доступа используется команда GRANT, а для их отзыва – команда REVOKE. В этих командах

указываются не только имена объектов, но также и имена ролей, которым предоставляются или у которых отзываются права доступа к объектам базы данных. Приведем примеры.

```
GRANT UPDATE ON students TO peter;
GRANT SELECT, INSERT, UPDATE ON progress TO PUBLIC;
REVOKE ALL ON progress FROM peter;
```

Ключевое слово ALL означает все права, а ключевое слово PUBLIC означает все роли.

Конкретной роли, например, user_one, может быть предоставлено членство в другой роли, например, user_two. Тем самым роль user_one получит все те привилегии, которые имеет роль user_two. Таким образом, роль может выполнять функции как отдельного пользователя, так и группы. Когда требуется предоставить какие-либо привилегии целой группе пользователей, сначала создают «групповую» роль, а затем предоставляют членство в этой «групповой» роли другим ролям (т. е. другим пользователям). Это членство в «групповой» роли можно также и отозвать.

```
GRANT group_role TO role1, ... ;
REVOKE group_role FROM role1, ... ;
```

3.4. Управление базами данных

Под управлением одного сервера может быть создано несколько баз данных. В свою очередь, база данных может содержать одну или более так называемых схем. Схема – это способ логического разделения объектов в базе данных. Поэтому в разных схемах одной и той же базы данных могут существовать одноименные объекты, например, таблицы. Получаем такую иерархию: сервер, база данных, схема, таблица (представление, функция, триггер и т. д.).

Для создания и удаления баз данных используются команды:

```
CREATE DATABASE имя_базы_данных;
DROP DATABASE имя_базы_данных;
```

Эти же действия можно выполнить из среды операционной системы, используя утилиты PostgreSQL:

```
/usr/local/pgsql/bin/createdb имя_базы_данных
/usr/local/pgsql/bin/dropdb имя_базы_данных
```

Новая база данных создается путем копирования содержимого системной базы данных с именем template1, которая является шаблоном. Поэтому если вы создадите дополнительные объекты в базе данных template1, то при создании новой базы данных эти объекты будут сразу появляться в ней. Существует еще одна стандартная база данных – template0. Она содержит те же объекты, что и первоначальная версия базы данных template1. Но базу данных template0 нельзя изменять, потому что она служит для выполнения ряда операций с базами данных. Например, восстанавливать ранее сохраненные данные из другой базы данных следует только в базу данных, созданную на основе template0. Для создания базы данных на основе template0 поступают так:

```
CREATE DATABASE имя_базы_данных TEMPLATE template0;
```

или так

```
/usr/local/pgsql/bin/createdb -T template0 имя_базы_данных
```

В процессе установки СУБД PostgreSQL создается база данных с именем postgres, которая является просто копией template1. База данных postgres является базой данных по умолчанию и может быть удалена, а затем создана снова по вашему желанию.

Посмотреть список баз данных можно из утилиты psql с помощью команды (строчная буква «L»)

```
\l
```

Для создания и удаления схем используются команды:

```
CREATE SCHEMA myschema;
DROP SCHEMA myschema CASCADE;
```

Ключевое слово CASCADE означает, что будет удалена схема со всеми объектами, созданными в ней.

По умолчанию в каждой вновь создаваемой базе данных есть схема public, в которой размещаются объекты базы данных, создаваемые без указания конкретной схемы. Но если нужно создать таблицу в конкретной схеме, то делают так:

```
CREATE TABLE myschema.mytable ( ...);
```

А если требуется предоставить привилегии доступа к этой таблице другим пользователям, то сначала нужно предоставить привилегию доступа к самой схеме, а затем уже к конкретному объекту, в данном случае к таблице:

```
GRANT USAGE ON SCHEMA имя_схемы TO имя_пользователя;
GRANT SELECT, INSERT ON имя_схемы.имя_таблицы TO
имя_пользователя;
```

Посмотреть список схем в базе данных можно из утилиты psql с помощью команды

```
\dn
```

А эта команда выведет дополнительно список системных схем с расширенной информацией о них:

```
\dnS+
```

3.5. Обслуживание базы данных

Периодическое выполнение процедур обслуживание базы данных направлено на повышение быстродействия и надежности функционирования всей системы. Одной из таких процедур является так называемое «вакуумирование» (vacuuming). Эта процедура предназначена для освобождения неиспользуемого дискового пространства в таблицах базы данных. Такое пространство образуется в процессе модифицирования таблиц, когда в них присутствуют различные версии одних и тех же строк, необходимые для реализации транзакций. При удалении строки из таблицы не происходит мгновенного освобождения занимаемого ею места и возвращения его операционной системе. Для этого необходима специальная процедура, которой и является «вакуумирование». Она может выполняться как в ручном режиме, так и в автоматическом. Сразу после установки сервера баз данных его конфигурация такова, что эта процедура выполняется автоматически в определенные моменты времени. Для отключения такого режима нужно параметру autovacuum в файле postgresql.conf дать значение off и перезапустить сервер. Если вы посмотрите список выполняющихся процессов с помощью команды ps до и после отключения этого параметра, то увидите, что процесс с описанием «postgres: autovacuum launcher process» исчезнет из списка.

Выполнять «вакуумирование» можно вручную с помощью команды VACUUM. Она имеет много параметров, позволяющих выполнять обработку конкретных таблиц и даже столбцов.

Важным вопросом является периодическое обновление статистики, которая собирается планировщиком запросов и служит для формирования плана выполнения запросов к базе данных. Если статистика устарела, то планировщик может сформировать неоптимальный план выполнения запроса, который будет выполняться дольше. При автоматическом «вакуумировании» это обновление статистики также выполняется. А в ручном режиме необходимо использовать команду ANALYZE, например, из среды утилиты psql.

Если в вашей базе данных используются индексы, то имеет смысл периодически перестраивать все индексы с помощью команды REINDEX. Индексы, построенные на основе B-дерева, работают быстрее, когда значения, близкие логически, расположены в физически смежных страницах дискового пространства.

В процессе своей работы сервер баз данных формирует диагностические сообщения в случае возникновения сбоев или нештатных ситуаций. Эти сообщения могут оказаться полезными для последующего анализа этих ситуаций. Поэтому целесообразно направлять все такие сообщения в специальный файл-журнал. В гл. 1 нашего пособия описан порядок запуска сервера. В команде присутствует параметр -l logfile, который указывает имя файла-журнала. В нашем случае этот файл так и назывался – logfile. Поскольку данный файл увеличивается в размерах со временем, его нужно периодически обнулять, сохраняя старую версию в архиве. Конечно, в случае когда для проведения обслуживания вашего сервера допускается его остановить, можно воспользоваться этим моментом для обновления файла-журнала. Однако если останавливать сервер нельзя, надо прибегнуть к помощи специальных средств, работающих с файлами-журналами и выполняющими ротацию файлов-журналов «на ходу».

Поскольку в базе данных самое ценное – это сами данные, то необходимо регулярно производить резервное копирование этих данных. PostgreSQL предлагает несколько способов выполнения этой процедуры. Мы рассмотрим только один, а именно: сохранение данных в виде SQL-команд в текстовом файле. Идея заключается в том, что такой файл можно передать обратно серверу, и он восстановит базу данных в первоначальном виде. Для выполнения такого сохранения базы данных предназначена утилита pg_dump. Она имеет много параметров, изучить которые можно такими способами:

```
/usr/local/pgsql/bin/pg_dump --help
man pg_dump
```

Для выгрузки всей базы данных, например, my_db, в файл save_db.sql с явным указанием имен столбцов таблиц в командах INSERT выполните команду:

```
/usr/local/pgsql/bin/pg_dump --column-inserts -f save_db.sql
my_db
```

Нужно учитывать, что, как и все утилиты PostgreSQL, pg_dump по умолчанию подключается к базе данных с правами пользователя базы данных, имя которого совпадает с именем пользователя операционной системы. Если вам нужно указать другое имя, используйте параметр -U. Например:

```
/usr/local/pgsql/bin/pg_dump -U user_name…….
```

Сохраненную базу данных можно восстановить с помощью утилиты psql. При этом сначала нужно создать пустую целевую базу данных, поскольку psql ее не создает. Также должны существовать учетные записи всех пользователей, которые владеют объектами, представленными в базе данных, сохраненной в файле, например, save_db.sql. Когда целевая база данных, например, my_db, создана, выполним команду:

```
/usr/local/pgsql/bin/psql -d my_db -f save_db.sql -U postgres
```

Имя пользователя может и не указываться, если имя пользователя по умолчанию вас устраивает.

Утилита pg_dump делает резервную копию только одной базы данных за раз. Для создания резервной копии всех баз данных, обслуживаемых данным сервером, служит утилита pg_dumpall. Она формирует также и команды для создания впоследствии учетных записей пользователей.

Важной задачей администратора является мониторинг активности сервера баз данных, т. е. получение информации о том, чем сейчас занят сервер, каково его текущее состояние. Для выполнения мониторинга используются не только средства самой СУБД PostgreSQL, но также и обычные утилиты операционной системы UNIX. Одна из таких утилит – ps, позволяющая увидеть список выполняющихся процессов. Чтобы увидеть только процессы, владельцем которых является пользователь postgres, нужно сделать так:

```
ps -auxww | grep "^postgres"
```

Можно использовать и более простой вариант этой команды:

```
ps -ax | grep postgres | grep -v grep
```

Полезными могут оказаться также утилиты top и vmstat.

В PostgreSQL существует подсистема сбора статистики, которая накапливает разнообразную информацию о работе сервера, в том числе: число обращений к таблицам и индексам, число строк в таблицах, результаты выполнения команд VACUUM и ANALYZE для каждой таблицы и т. д. Поскольку сбор статистической информации замедляет работу сервера, в файле postgresql.conf предусмотрены параметры для отмены выполнения этой работы. Для просмотра собранной информации можно использовать многочисленные системные представления (views), описанные в разделе «The Statistics Collector». Однако нужно учитывать, что данные в этих представлениях не обновляются мгновенно. Например, пока текущая транзакция не завершится, ее выполнение не влияет на итоговые статистические результаты. В качестве примера покажем, как можно увидеть статистику, собранную для таблиц, созданных пользователем (есть еще и системные таблицы):

```
SELECT * FROM pg_statio_user_tables;
```

Узнать, как создано это представление, можно с помощью утилиты psql:

```
\d+  pg_statio_user_tables
```

Коротко расскажем о блокировках. Они могут возникать при параллельном выполнении транзакций, которые конкурируют за ресурсы базы данных, в том числе таблицы и индексы. С помощью системного представления (view) pg_locks можно выявить неснятые блокировки таблиц в конкретной базе данных. Это позволит выявить конкурирующие транзакции и поможет определить «узкие» места в базе данных. В качестве примера предлагаем выполнить следующий запрос:

```
SELECT l.virtualtransaction, l.locktype, l.granted, c.relname,
d.datname, c.relkind FROM pg_locks l, pg_class c, pg_database
d WHERE l.relation = c.oid AND l.database = d.oid;
```

В этом запросе используются также системные таблицы pg_class и pg_database. В столбце virtualtransaction будет выведен виртуальный ID транзакции, которая удерживает блокировку или ожидает ее. Столбец locktype содержит тип блокируемого объекта (например, relation – таблица). В столбце granted выводится значение «t», если блокировка получена и удерживается, или «f», если она ожидается. Столбец relname представляет имя объекта базы данных, например имя таблицы, а столбец datname – имя базы данных. И наконец, столбец relkind отражает вид блокируемого объекта: r – обычная таблица, i – индекс, v – представление и т. д.

Контрольные вопросы и задания

1. Какие конфигурационные файлы сервера баз данных вы знаете?

2. С помощью утилиты psql посмотрите в системной таблице pg_settings, какие единицы измерения используются по умолчанию для задания параметров сервера, а также какие значения могут присваиваться параметрам, использующим перечислимый тип данных. Для ознакомления со структурой этой таблицы воспользуйтесь документацией на PostgreSQL, глава «System Catalogs». Можно использовать и команду

```
\d pg_settings
```

3. В главе «Server Configuration» документации на PostgreSQL представлены все параметры сервера баз данных. Ознакомьтесь с ними и попробуйте выполнить изменение значений параметров, используя все способы, описанные в тексте этой главы пособия.

4. Самостоятельно ознакомьтесь с примерами конфигураций, приведенными в разделе «Client Authentication» документации на PostgreSQL. Организуйте проверку подлинности клиента по паролю при подключении его к серверу по сети с конкретного IP-адреса.

5. Использовать «групповые» привилегии можно одним из двух способов: с помощью команды SET ROLE или с помощью механизма наследования привилегий. Изучите эти вопросы самостоятельно по технической документации на PostgreSQL, глава «Database Roles». Смоделируйте ситуацию, в которой было бы оправданно использование «групповых» ролей. Создайте таблицу базы данных и необходимые роли.

6. Кроме привилегий SELECT, INSERT, UPDATE и DELETE есть еще целый ряд других привилегий. Изучите эти вопросы

самостоятельно по технической документации на PostgreSQL, описание команды GRANT.

7. Создайте базу данных и схему в этой базе данных, а в этой схеме – таблицу. Предоставьте привилегии доступа к этой таблице какому-нибудь пользователю.

8. Самостоятельно ознакомьтесь с понятием табличного пространства, представленным в разделе «Tablespaces» документации на PostgreSQL. Посмотрите описание команды CREATE TABLESPACE.

9. Самостоятельно ознакомьтесь с технологиями поддержки параметров локализации, применяемыми в СУБД PostgreSQL. Обратитесь к разделу документации «Localization».

10. Отмените автоматическое «вакуумирование» баз данных с помощью конфигурационного файла postgresql.conf и убедитесь, что ваши действия дали эффект.

11. Самостоятельно ознакомьтесь с параметрами сервера, которые отвечают за формирование файла-журнала: его имя, программа для ротации этих файлов, уровни серьезности журналируемых событий, формат сообщений и т. д. Обратитесь к разделу документации «Error Reporting and Logging».

12. Используя утилиту pg_dump, сохраните какую-нибудь базу данных по вашему выбору. При этом один раз используйте параметр --column-inserts, а в другой раз не указывайте этот параметр. Сравните полученные результаты, изучив содержимое команд INSERT в полученных результирующих файлах.

13. Утилита pg_dump позволяет сохранить не только всю базу данных, но также и ее отдельные таблицы, указав их имена с помощью параметра -t. Выполните сохранение одной-двух таблиц.

14. Самостоятельно ознакомьтесь с системными представлениями (views), содержащими собранную статистику о работе сервера. Обратитесь к разделу «The Statistics Collector» документации.

15. С целью изучения механизма блокировок создайте две или три параллельные транзакции, конкурирующие за одну и ту же таблицу в целом либо за ее отдельные строки. Как это сделать, описано в п. 2.4 «Управление транзакциями» нашего пособия. Затем на другом терминале запустите утилиту psql и с помощью запроса к системному представлению pg_locks, приведенного в этом разделе пособия, посмотрите состояние блокировок в базе данных. Для получения более точного представления о том, как работает механизм блокировок, начинайте и завершайте транзакции, чтобы они то запрашивали ресурсы, то освобождали их. При этом после каждого изменения конфигурации транзакций выполняйте запрос к системному представлению pg_locks.

4. ПРОГРАММИРОВАНИЕ СЕРВЕРНОЙ ЧАСТИ СИСТЕМЫ УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ

4.1. Функции

При разработке приложений с базами данных часто бывает целесообразно переложить часть операций с данными с клиентского приложения на серверную часть СУБД. Это позволяет упростить программный код приложения, уменьшить объем данных, передаваемых по сети и ускорить работу приложения.

Для расширения возможностей СУБД разрабатываются так называемые хранимые процедуры, которые в рамках PostgreSQL называются функциями. Такие функции можно писать на языке SQL. В функцию можно включать произвольные SQL-команды. Возвращаемым значением будет первая строка результата выполнения последней SQL-команды. Можно использовать не только оператор SELECT, но также INSERT, UPDATE и DELETE. Если функция не должна возвращать никакого полезного значения, тогда возвращаемым типом будет void. Функция может получать аргументы. Эти аргументы могут иметь модификаторы IN и OUT. Первый из них используется по умолчанию и означает параметр, значение которого будет изменяться внутри функции, но вне функции его новое значение не будет доступно. Параметр с ключевым словом OUT используется для того, чтобы возвратить из функции более одного значения. Парные знаки $$ ограничивают непосредственно тело (определение) функции. Завершает определение функции наименование языка, на котором она написана. Например:

```
CREATE FUNCTION add_numbers( x integer, y integer ) RETURNS
integer AS $$
  SELECT x + y;
$$ LANGUAGE SQL;

ais=> select * from add_numbers( 2, 4 );
 add_numbers
-------------
           6
(1 строка)
```

Вместо имен параметров можно использовать их номера:

```
CREATE FUNCTION add_numbers2( integer, integer ) RETURNS inte-
ger AS $$
  SELECT $1 + $2;
```

```
$$ LANGUAGE SQL;
```

А это пример использования параметров с ключевым словом OUT:

```
CREATE FUNCTION sum_and_product( x int, y int, OUT sum int,
                                 OUT product int )
AS $$
  SELECT x + y, x * y;
$$ LANGUAGE SQL;

ais=> select * from sum_and_product( 2, 6 );
 sum | product
-----+---------
   8 |      12
(1 строка)
```

Вызов функции можно использовать в подзапросе в предложении FROM, но в таком случае требуется использовать псевдоним (мы взяли my_data):

```
ais=> select sum + product as total_number from (select * from
sum_and_product( 2, 6 )) as my_data;
 total_number
--------------
           20
(1 строка)
```

Для удаления функции используется команда DROP FUNCTION. Например:

```
DROP FUNCTION sum_and_product( int, int );
```

Обратите внимание, что в скобках указываются только типы данных без указания имен параметров. При этом параметры, объявленные с ключевым словом OUT, можно не указывать.

Посмотреть список всех функций, созданных в вашей базе данных, можно с помощью интерактивного терминала psql. Сначала включим расширенный вывод, поскольку он более удобен для просмотра данных, которые не помещаются на одной строке.

```
ais=> \x
Расширенный вывод включен.

ais=> \dfn

Список функций
```

```
-[ RECORD 1 ]----------+--------------------------------------
Схема                  | public
Имя                    | add_numbers
Тип данных результата  | integer
Типы данных аргументов | x integer, y integer
Тип                    | обычная
-[ RECORD 2 ]----------+--------------------------------------
----
Схема                  | public
Имя                    | add_numbers2
Тип данных результата  | integer
Типы данных аргументов | integer, integer
Тип                    | обычная
```

Функции могут возвращать и табличные значения. В качестве примера обратимся к той базе данных, которая была представлена в гл. 2. Напишем функцию, подсчитывающую количество фамилий в таблице students («Студенты»), начинающихся на каждую букву. Однако если в таблице нет ни одной фамилии, начинающейся, скажем, с буквы «Ш», то в выводе функции эта буква представлена не будет. Выражение RETURNS TABLE(letter char(1), num bigint) означает, что формируемая таблица будет иметь два поля (столбца) с указанными именами и типами данных. Выражение OR REPLACE означает, что если функция уже существует, то она будет удалена и заменена новой версией.

В качестве примера напишем функцию, которая будет подсчитывать число студентов, имеющих фамилии, начинающиеся на каждую букву алфавита. При этом учитываются только буквы алфавита, фактически представленные в таблице students: если, к примеру, отсутствуют фамилии, начинающиеся с буквы «Ш», то эта функция просто не включает такую букву в результирующий вывод.

```
CREATE OR REPLACE FUNCTION count_letters()
  RETURNS TABLE( letter char( 1 ), num bigint ) AS
$$
  SELECT substr( name, 1, 1 ) AS letter, count( * )
  FROM students GROUP BY letter ORDER BY letter;
$$ LANGUAGE SQL;
```

4.2. Триггеры

Триггер – это механизм, заставляющий СУБД выполнить конкретную функцию, когда выполняется определенный тип операций.

Триггеры могут быть связаны с таблицами и представлениями (views).

Триггеры, связанные с таблицами, могут выполняться как ДО (BEFORE), так и ПОСЛЕ (AFTER) операций INSERT, UPDATE, DELETE. При этом возможно выполнение триггера либо для каждой модифицированной строки таблицы, либо однократно для SQL-команды в целом. Если случается конкретное событие, приводящее к срабатыванию триггера, то вызывается так называемая триггерная функция, которая и обрабатывает это событие.

Триггеры, связанные с представлениями (views), выполняются ВМЕСТО операций INSERT, UPDATE, DELETE.

Для создания триггера сначала нужно создать триггерную функцию. Эта функция не должна принимать никаких аргументов и должна возвращать значение типа trigger. Необходимые ей данные триггерная функция получает от СУБД без участия программиста.

Триггеры могут быть двух типов с точки зрения числа повторных вызовов триггерной функции:

– функция может вызываться для каждой строки, на которую влияет (affected) команда, вызвавшая срабатывание триггера;

– функция может вызываться только один раз, независимо от числа строк, подвергшихся воздействию команды, вызвавшей срабатывание триггера. Даже если таких строк не будет ни одной, триггерная функция вызывается все равно.

Триггеры первого типа называют триггерами уровня строки (row-level), а триггеры второго типа – триггерами уровня команды (statement-level).

Триггерные функции, вызываемые триггерами уровня SQL-команды, должны всегда возвращать значение NULL. Триггерные функции, вызываемые триггерами уровня строки (row-level), могут возвращать строку таблицы, если это необходимо с точки зрения логики этой функции.

Триггер уровня строки, выполняемый ДО операции, может принимать одно из двух решений:

– он может возвратить значение NULL, чтобы предотвратить операцию с текущей строкой таблицы, для которой и вызван этот триггер (например, вставку, обновление или удаление строки таблицы);

– при выполнении операций вставки или обновления триггер уровня строки может модифицировать вставляемую или обновляемую строку таблицы, поскольку именно строка, возвращаемая триггером, и будет вставлена в таблицу или обновлена в ней.

Поэтому триггер такого типа, если не планируется отмена операции или модифицирование строки таблицы, должен возвратить неизмененную строку таблицы. В случае операции вставки (INSERT) и обновления (UPDATE) это будет специальное значение NEW, а в случае операции удаления строки (DELETE) это будет специальное значение OLD.

Для триггеров уровня строки, выполняемых после операции, значение, возвращаемое триггерной функцией, просто игнорируется, поэтому они могут возвращать значение NULL.

Если для одного и того же события, имеющего место в отношении данной таблицы, определено несколько триггеров, то они срабатывают в алфавитном порядке их имен. Если предшествующий BEFORE-триггер модифицирует строку таблицы, то последующий BEFORE-триггер получает на вход уже модифицированную строку. Если предыдущий BEFORE-триггер возвращает значение NULL, то операция над данной строкой таблицы отменяется и последующие BEFORE-триггеры не срабатывают.

В типичном случае BEFORE-триггеры уровня строки используются для проверки или модифицирования данных, которые будут вставлены в таблицу или обновлены в ней. Например, такой триггер может использоваться для вставки значения текущего времени в поле типа timestamp или для проверки согласованности значений двух или более полей текущей строки таблицы.

AFTER-триггеры логично использовать для продвижения изменений, сделанных в текущей таблице, в другие таблицы или выполнять проверки согласованности данных с другими таблицами.

Упрощенный синтаксис команды для создания триггера таков:

```
CREATE TRIGGER имя_триггера { BEFORE | AFTER | INSTEAD OF } {
событие [ OR ... ] }
  ON имя_таблицы
  [ FOR [ EACH ] { ROW | STATEMENT } ]
  EXECUTE PROCEDURE имя_функции ( аргументы )

где «событие» – это одно из:

  INSERT
  UPDATE [ OF имя_столбца [, ... ] ]
  DELETE
  TRUNCATE
```

4.3. Язык PL/pgSQL

PL/pgSQL – это процедурный язык СУБД PostgreSQL. Он может использоваться для создания обычных функций и триггерных функций. Этот язык позволяет дополнить язык SQL управляющими структурами. С его помощью можно выполнять сложные вычисления. Функции, написанные на этом языке, могут использоваться везде, где могли бы использоваться встроенные функции языка SQL, например, в индексных выражениях при создании индексов.

Данный язык позволяет повысить эффективность работы приложения с базой данных за счет того, что в рамках одной процедуры, написанной на этом языке, могут быть сгруппированы несколько SQL-операторов, которые хранятся на сервере. Поэтому клиентскому приложению не требуется выполнять эти SQL-операторы по одному, организуя каждый раз взаимодействие с сервером и тем самым увеличивая сетевой трафик. Также не выполняется передача промежуточных результатов вычислений от сервера к клиенту, тем самым также сокращается число взаимодействий клиента и сервера, что позволяет ускорить обработку данных. Ведь значительная ее часть будет выполняться на сервере при выполнении хранимых процедур, а клиент получит уже окончательные результаты.

Функции на языке PL/pgSQL оформляются в виде блоков (в квадратных скобках указаны необязательные элементы):

```
[ <<метка>> ]
[ DECLARE
  объявления ]
BEGIN
  операторы
END [ метка ];
```

Внутри блока могут содержаться вложенные блоки, которые удобно использовать для отражения логической структуры функции. Переменные, объявленные во вложенном блоке, скрывают одноименные переменные, объявленные во внешнем блоке.

Все ключевые слова являются нечувствительными к регистру символов, поэтому их можно вводить как в верхнем, так и в нижнем регистре.

Приведем пример функции, представленный в документации на PostgreSQL, дополнив его комментариями.

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
  -- Объявим переменную типа integer и инициализируем ее.
  quantity integer := 30;
BEGIN
  -- Этот оператор выведет сообщение, в котором вместо знака %
  -- будет подставлено значение переменной quantity, равное
30.
  RAISE NOTICE 'Quantity here is %', quantity;
  quantity := 50;  -- присвоим переменной новое значение
  --
  -- Создадим вложенный блок.
  --
  DECLARE
    -- Объявим переменную типа integer и инициализируем ее.
    -- Имя этой переменой такое же, как и переменной в главном
    -- блоке.
    quantity integer := 80;
  BEGIN
    -- Этот оператор выведет значение 80.
    RAISE NOTICE 'Quantity here is %', quantity;

    -- Этот оператор выведет значение 50. Поскольку имени
    -- переменной предшествует имя метки внешнего блока, будет
    -- использована переменная quantity из внешнего блока
    -- outerblock.
    RAISE NOTICE 'Outer quantity here is %',
outerblock.quantity;
  END;
  -- Вложенный блок завершился, значит, эта команда выведет
  -- значение переменной, объявленной в главном блоке, т. е. 50.
  RAISE NOTICE 'Quantity here is %', quantity;

  RETURN quantity; -- возвратим результат
END;
$$ LANGUAGE plpgsql;
```

Как и при написании функций на языке SQL, парные символы «$» служат в качестве ограничителей определения функции. Но в операторе LANGUAGE указывается не SQL, а plpgsql.

Управляющие конструкции языка PL/pgSQL очень похожи на аналогичные конструкции других языков программирования. Это касается условных операторов, операторов для организации циклов, операторов завершения процедуры (RETURN).

В случае возникновения ошибки при выполнении функции PL/pgSQL работа функции прерывается. Но можно перехватывать возникающие ошибки и обрабатывать их тем или иным образом. Для этого в блок BEGIN…END вводится ключевое слово EXCEPTION.

```
[ <<label>> ]
[ DECLARE
  объявления ]
BEGIN
  операторы
EXCEPTION
  WHEN условие [ OR условие ... ] THEN
    операторы_для_обработки_ошибки
  [ WHEN условие [ OR условие ... ] THEN
      операторы_для_обработки_ошибки
. ]
END;
```

Все условия, используемые в блоке обработки ошибок, имеют стандартизированные имена, приведенные в приложении А к документации на PostgreSQL. Если для возникшей ошибки предусмотрено соответствующее условие в данном блоке BEGIN..END, тогда эта ошибка обрабатывается здесь. Если же для нее обработчик не предусмотрен в данном блоке, тогда ошибка продвигается во внешний блок и обрабатывается там. Если же там обработчика для данной ошибки также нет, тогда выполнение функции прерывается.

Для вывода сообщений пользователю или для генерирования ошибок служит команда RAISE. Покажем один из вариантов ее синтаксиса.

```
RAISE [ уровень ] 'формат' [, выражение [, ... ]];
```

Здесь уровень означает степень серьезности сообщения: DEBUG, LOG, INFO, NOTICE, WARNING и EXCEPTION. По умолчанию используется EXCEPTION, что означает формирование ошибки. Параметр 'формат' служит для формирования текста сообщения, за этим параметром могут следовать переменные, значения которых подставляются в строку 'формат' в те позиции, которые обозначены символом «%». Приведем простой пример:

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

Теперь перейдем к такому важному компоненту любого языка программирования, как переменные. Переменные в языке PL/pgSQL

могут иметь любой тип данных, имеющийся в PostgreSQL, например integer, varchar и т. д.

```
-- DEFAULT означает присваивание :=
quantity integer DEFAULT 32;

-- строковое значение нужно заключить в одинарные кавычки
url varchar := 'http://mysite.com';

-- можно создать константу и инициализировать ее
user_id CONSTANT integer := 10;
```

Кроме того, предусмотрены дополнительные способы создания переменных. Часто требуются переменные, которые будут содержать значения каких-либо полей из таблицы базы данных. Например, если нам нужна переменная для хранения значения поля user_id из таблицы users, то можно воспользоваться следующим оператором:

```
user_id users.user_id%TYPE;
```

Такой оператор избавляет нас от необходимости знать тип данных этого поля. При изменении типа данных в таблице нам не придется корректировать код функции, в которой используется это поле.

Переменная так называемого строкового типа (row-type variable) может содержать всю строку таблицы базы данных. Такой тип данных является составным. Такая переменная объявляется следующим образом:

```
имя_переменной имя_таблицы%ROWTYPE;
```

Переменные такого типа могут использоваться в качестве параметров функций. В качестве иллюстрации воспользуемся примером, приведенным в документации на PostgreSQL, дополнив этот пример комментариями. В этой функции table1 и table2 – это имена существующих таблиц, t_row – это имя формального параметра функции, этот параметр будет содержать всю строку таблицы table1.

```
CREATE FUNCTION merge_fields( t_row table1 ) RETURNS text AS
$$
DECLARE
  -- Объявим переменную строкового типа для таблицы table2.
  t2_row table2%ROWTYPE;
BEGIN
  -- Выберем одну строку из таблицы table2 и запишем значения
  -- ее полей в переменную t2_row.
```

```
  SELECT * INTO t2_row FROM table2 WHERE ... ;
  -- Получим результирующее значение путем конкатенации значений
  -- полей из записей двух таблиц. Операция конкатенации «||».
  -- Для обращения к конкретному полю строковой переменной служит
  -- операция «.».
  RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

-- Так можно воспользоваться созданной функцией.
SELECT merge_fields( t.* ) FROM table1 t WHERE ... ;
```

Существуют переменные, похожие на строковые переменные. Они называются переменными типа записи (record variables) и объявляются так:

```
имя_переменной RECORD;
```

Переменные такого типа не имеют предопределенной структуры. Они ее получают только тогда, когда им присваивается содержимое строки, полученной в результате выполнения оператора SELECT. Далее в примерах вы увидите использование такого типа данных.

Результат SQL-команды, возвращающий точно одну строку, можно записать в переменную типа записи (record variable), строковую переменную (row-type variable) или группу скалярных переменных. Общий синтаксис таков:

```
SELECT запрос INTO [STRICT] целевые_переменные FROM ...;
```

Если указано ключевое слово STRICT, тогда запрос должен возвращать точно одну строку, иначе возникнет ошибка времени исполнения. Если же ключевое слово STRICT не указано, тогда в переменную будет записана первая возвращенная строка. Если не было возвращено ни одной строки, тогда в переменную будут записаны значения NULL. Но нужно помнить, что если в запросе не использовалось ключевое слово ORDER BY, тогда первая строка результата не является точно определенной. В следующем примере показан запрос и используется специальная переменная FOUND для проверки, была ли действительно выбрана строка в результате выполнения запроса.

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
  RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

А этот пример иллюстрирует использование ключевого слова STRICT.

```
BEGIN
  SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
EXCEPTION
  -- Если не выбрано ни одной строки, сформируем исключение.
  WHEN NO_DATA_FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
  -- Если выбрано более одной строки, сформируем исключение.
  WHEN TOO_MANY_ROWS THEN
    RAISE EXCEPTION 'employee % not unique', myname;
END;
```

Добавим, что команды UPDATE, INSERT и DELETE также устанавливают значение переменной FOUND равным «истине», если хотя бы одна строка в таблице была обновлена, вставлена или удалена соответственно.

Иногда бывает полезным пустой оператор – NULL, который может повысить наглядность программного кода. Например:

```
BEGIN
  y := x / 0;
EXCEPTION
  -- Если имело место деление на нуль, то управление ходом
  -- выполнения процедуры будет передано сюда.
  WHEN division_by_zero THEN
    NULL;  -- не делаем ничего, просто игнорируем ошибку
END;
```

В заключение этого параграфа приведем текст еще одной функции. Она предназначена для заполнения таблицы «Студенты» тестовыми данными.

```
CREATE OR REPLACE FUNCTION generate_students_data()
  RETURNS TABLE( mark_book numeric( 5 ), name text,
                 psp_ser numeric( 4 ), psp_num numeric( 6 ) )
AS
$$
BEGIN
  -- Создадим последовательности для полей mark_book, psp_ser,
  -- psp_num.

  -- Значения поля «Номер зачетной книжки» должны быть
  -- пятизначными.
```

```
  CREATE TEMP SEQUENCE New_mark_book START WITH 10000;

  -- Значения поля «Серия паспорта» должны быть четырехзначными.
  CREATE TEMP SEQUENCE New_psp_ser START WITH 1000;

  -- Значения поля «Номер паспорта» должны быть шестизначными.
  CREATE TEMP SEQUENCE New_psp_num START WITH 100000;
  RETURN QUERY
  SELECT NEXTVAL( 'New_mark_book' )::numeric( 5 ),
         lname_base || lname_suffix || ' ' ||
         fname || ' ' || pname,
         NEXTVAL( 'New_psp_ser' )::numeric( 4 ),
         NEXTVAL( 'New_psp_num' )::numeric( 6 )
  -- Формируем декартово произведение фрагментов фамилии,
  -- имени и отчества, т. е. комбинации каждого значения
  -- из каждой виртуальной таблицы с каждым значением из других
  -- виртуальных таблиц.
  FROM ( VALUES ( 'Логин' ), ( 'Перл' ),
                ( 'Программ' ), ( 'Дебаг' )
       ) AS lname_bases ( lname_base ) NATURAL JOIN
       ( VALUES ( 'ов' ), ( 'овых' ), ( 'овский' ),
                ( 'овичев' ), ( 'енко' )
       ) AS lname_suffixes ( lname_suffix ) NATURAL JOIN
       ( VALUES ( 'Андрей' ), ( 'Иван' ), ( 'Николай' ),
                ( 'Петр' )
       ) AS fnames ( fname ) NATURAL JOIN
       ( VALUES ( 'Анатольевич' ), ( 'Кириллович' ),
                ( 'Павлович' ), ( 'Тихонович' ) )
           AS pnames ( pname );

  -- Удалим врЕменные последовательности.
  DROP SEQUENCE New_mark_book;
  DROP SEQUENCE New_psp_ser;
  DROP SEQUENCE New_psp_num;
END
$$ LANGUAGE plpgsql;
```

Использовать эту функцию для непосредственного ввода данных в таблицу Students нужно так:

```
INSERT INTO students SELECT * FROM generate_students_data();
```

4.4. Хранение иерархических структур данных в базах данных реляционного типа

Проиллюстрировать возможности языка PL/pgSQL целесообразно на каком-то масштабном примере, когда можно было бы использовать как функции, так и триггеры. В качестве такого примера изберем хра-

нение иерархических структур в базах данных реляционного типа. Обработка иерархических структур, хранящихся в реляционных базах данных, требует выполнения разнообразных операций над реляционными таблицами. Зачастую одна операция над иерархией требует выполнения целой группы элементарных SQL-команд. В таких случаях было бы очень удобно объединить эти команды в целостную процедуру, написать которую целесообразно на языке PL/pgSQL.

В настоящее время предложено несколько подходов к организации иерархий в реляционных базах данных. Мы рассмотрим только один из методов, который построен на основе так называемых списков смежности (adjacency list). За основу изложения материала принята книга Joe Celko «Joe Celko's Trees and hierarchies in SQL for smarties». Однако были исправлены опечатки и ошибки в некоторых процедурах, внесены дополнительные операторы, а также добавлены дополнительные комментарии.

Для того чтобы изучать различные способы и технологии расширения возможностей сервера баз данных, нужно сначала подготовить учебную базу данных и создать в ней ряд таблиц. Первая команда – создание базы данных. Выполнять ее нужно, войдя в систему с правами пользователя postgres.

```
createdb ais
```

В качестве имени базы данных мы выбрали аббревиатуру от Administering of information systems (ais), в качестве предметной области – организационную иерархию должностей работников. Поскольку этот пример – учебный, то в базе данных всего две реляционные таблицы. Это значительное упрощение реальной жизни.

```
-- ------------------------------------------------------------
----
-- Таблица "Персонал"
-- ------------------------------------------------------------
----
DROP TABLE IF EXISTS Personnel CASCADE;

CREATE TABLE Personnel
( emp_nbr INTEGER                    -- код работника
    DEFAULT 0 NOT NULL PRIMARY KEY,
  emp_name VARCHAR( 10 )             -- имя работника
    DEFAULT '{ {vacant} }' NOT NULL,
  address VARCHAR( 35 ) NOT NULL,    -- адрес работника
  birth_date DATE NOT NULL           -- день рождения работника
);
```

```
-- Произведем первоначальное заполнение таблицы.
INSERT INTO Personnel VALUES
( 0, 'вакансия', '',                          '2014-05-19' ),
( 1, 'Иван',     'ул. Любителей языка C',     '1962-12-01' ),
( 2, 'Петр',     'ул. UNIX гуру',             '1965-10-21' ),
( 3, 'Антон',    'ул. Ассемблерная',          '1964-04-17' ),
( 4, 'Захар',    'ул. им. СУБД PostgreSQL',   '1963-09-27' ),
( 5, 'Ирина',    'просп. Программистов',      '1968-05-12' ),
( 6, 'Анна',     'пер. Перловый',             '1969-03-20' ),
( 7, 'Андрей',   'пл. Баз данных',            '1945-11-07' ),
( 8, 'Николай',  'наб. ОС Linux',             '1944-12-01' );

-- ------------------------------------------------------------
-- Таблица "Организационная структура"
-- ------------------------------------------------------------
DROP TABLE IF EXISTS Org_chart CASCADE;

CREATE TABLE Org_chart
( job_title VARCHAR( 30 )  -- наименование должности
    NOT NULL PRIMARY KEY,
  emp_nbr INTEGER          -- код работника
    DEFAULT 0 NOT NULL     -- 0 означает вакантную должность
    REFERENCES Personnel( emp_nbr )    -- внешний ключ
      ON DELETE SET DEFAULT
      ON UPDATE CASCADE
      -- Это ограничение будет отключаться при выполнении
      -- одной из хранимых процедур, поэтому DEFERRABLE.
      DEFERRABLE,
  boss_emp_nbr INTEGER     -- код начальника данного работника
    DEFAULT 0
    -- Поскольку null означает корень иерархии, то ограничение
    -- NOT NULL вводить не будем.
    REFERENCES Personnel( emp_nbr )    -- внешний ключ
      ON DELETE SET DEFAULT
      ON UPDATE CASCADE
      -- Это ограничение будет отключаться при выполнении
      -- одной из хранимых процедур, поэтому DEFERRABLE.
      DEFERRABLE,
  salary DECIMAL( 12, 4 )  -- зарплата работника, занимающего
                           -- эту должность
    NOT NULL CHECK ( salary >= 0.00 ),
  -- Работник не может быть сам себе начальником,
  -- т. е. код работника не должен совпадать с кодом
  -- начальника);
  -- если должность не занята, то код работника и код
  -- начальника равны 0.
  CHECK ( ( boss_emp_nbr <> emp_nbr ) OR
          ( boss_emp_nbr = 0 AND emp_nbr = 0 )
        ),
  -- Без этого внешнего ключа возможна ситуация, когда
```

```
  -- удаляется запись, значение поля emp_nbr которой
  -- используется в качестве значения поля boss_emp_nbr
  -- в других записях. Другими словами, работник, которого
  -- нет в орг. структуре, является начальником других
  -- работников, присутствующих в орг. структуре).
  FOREIGN KEY ( boss_emp_nbr )
    REFERENCES Org_chart ( emp_nbr )
    ON DELETE SET DEFAULT
    ON UPDATE CASCADE
    DEFERRABLE,

  -- Пришлось добавить и это ограничение, иначе
  -- внешний ключ FOREIGN KEY ( boss_emp_nbr ) создать
  -- невозможно.
  UNIQUE ( emp_nbr )
);

-- Произведем первоначальное заполнение таблицы.
-- Обратите внимание, что у главы компании нет начальника,
-- поэтому значение NULL.
INSERT INTO Org_chart VALUES
( 'Президент',           1, NULL, 1000.00 ),
( 'Вице-президент 1',    2, 1,    900.00 ),
( 'Вице-президент 2',    3, 1,    800.00 ),
( 'Архитектор',          4, 3,    700.00 ),
( 'Ведущий программист', 5, 3,    600.00 ),
( 'Программист C',       6, 3,    500.00 ),
( 'Программист Perl',    7, 5,    450.00 ),
( 'Оператор',            8, 5,    400.00 );
```

Создав таблицы и заполнив их начальными данными, можем приступить к созданию триггера для обеспечения целостности нашей базы данных. В соответствии с требованиями документации, сначала нужно создать триггерную функцию для нашего триггера. Эта функция возвращает значение типа trigger.

```
CREATE OR REPLACE FUNCTION check_org_chart() RETURNS trigger
AS
$$
BEGIN
  -- Эта функция только не позволяет сделать две и более
  -- записи с идентификатором босса, равным null, но против
  -- зацикливаний она бессильна.

  -- Это условие не позволяет удалить из таблицы последнюю
  -- запись (главный босс должен быть всегда -- это так и
  -- нужно?).
  -- ПРИМЕЧАНИЕ. Условие COUNT( boss_emp_nbr ) подсчитывает
  -- только те записи, у которых значение поля boss_emp_nbr
```

```
  -- не равно NULL. Условие COUNT( * ) подсчитывает все записи.
  IF ( SELECT COUNT( * ) FROM Org_chart ) - 1 <>
     ( SELECT COUNT( boss_emp_nbr ) FROM Org_chart )
  THEN
    -- Прервем выполнение функции и выведем сообщение.
    RAISE EXCEPTION 'Bad orgchart structure';
  ELSE
    -- В зависимости от вида операции (встроенная переменная
    -- TG_OP) с таблицей возвратим либо старую (OLD), либо
    -- новую (NEW) версии строки таблицы.
    IF ( TG_OP = 'DELETE' ) THEN
      RETURN OLD;
    ELSIF ( TG_OP = 'UPDATE' ) THEN
      RETURN NEW;
    ELSIF ( TG_OP = 'INSERT' ) THEN
      RETURN NEW;
    END IF;
    RETURN NULL;
  END IF;
END;
$$
LANGUAGE plpgsql;
```

Создадим триггер, который использует триггерную функцию check_org_chart(). Если такой триггер уже есть в базе данных, то предварительно удалим его.

```
DROP TRIGGER IF EXISTS check_org_chart ON Org_chart;
```

Триггер выполняется после каждой операции с таблицей, на это указывает ключевое слово AFTER. Он выполняется для КАЖДОЙ обрабатываемой строки таблицы, о чем говорит выражение FOR EACH ROW.

```
CREATE TRIGGER check_org_chart
AFTER INSERT OR UPDATE OR DELETE ON Org_chart
  FOR EACH ROW EXECUTE PROCEDURE check_org_chart();
```

Теперь мы можем приступить к созданию функций для управления иерархией. Важным этапом является проверка иерархии на предмет наличия циклов. Циклы могут быть короткие и длинные. Короткий цикл образуется, когда у работника А начальником является работник Б, а у работника Б – работник А. В длинном цикле большее число участников. Схематически длинный цикл можно показать так: А → Б → В → А. В этом цикле три участника. Следующая функция позволяет выполнить такую проверку. В качестве результата провер-

ки она возвращает символьное значение: «Tree» (циклов нет) или «Cycles» (циклы есть).

```
-- ------------------------------------------------------------
-- Функция для проверки структуры дерева на предмет
-- отсутствия циклов.
-- ------------------------------------------------------------
CREATE OR REPLACE FUNCTION tree_test() RETURNS CHAR( 6 ) AS
$$
BEGIN
  -- Создадим временную таблицу на основе иерархии
  -- должностей.
  CREATE TEMP TABLE Tree ON COMMIT DROP AS
    SELECT emp_nbr, boss_emp_nbr FROM Org_chart;

  -- Удаляем листья дерева. Условие означает следующее:
  -- SELECT COUNT( * ) -- общее фактическое число узлов
  -- дерева, т. к. каждая запись в таблице соответствует
  -- одному узлу;
  -- SELECT COUNT( * ) - 1 -- теоретическое число ребер
  -- дерева, которое должно быть на 1 меньше фактического
  -- числа узлов;
  -- SELECT COUNT( boss_emp_nbr ) -- число записей,
  -- для которых значение поля boss_emp_nbr не равно NULL
  -- (см. описание функции COUNT()), должно быть ровно на 1
  -- меньше, чем COUNT( * ), т. к. только у одного работника
  -- значение поля boss_emp_nbr может быть равно NULL --
  -- это глава организации.
  WHILE ( SELECT COUNT( * ) FROM Tree ) - 1
          = ( SELECT COUNT( boss_emp_nbr ) FROM Tree )
  LOOP
    -- Удаляем записи (строки) о работниках, которые
    -- не являются начальниками ни для одного из других
    -- работников (в подзапросе выбираются все начальники).
    DELETE
    FROM Tree
    WHERE Tree.emp_nbr NOT IN ( SELECT T2.boss_emp_nbr
                                FROM Tree AS T2
                                WHERE T2.boss_emp_nbr
                                      IS NOT NULL );
  END LOOP;

  -- Эта проверка должна выполняться уже после завершения
  -- удаления записей из таблицы Tree. Если записей
  -- не осталось, значит, дерево связанное.
  IF NOT EXISTS ( SELECT * FROM Tree )
  THEN
    RETURN ( 'Tree' );
  -- Если хоть одна запись осталась, значит,
  -- в дереве есть циклы.
```

```
  ELSE
    RETURN ( 'Cycles' );
  END IF;
END;
$$
LANGUAGE plpgsql;
```

Один из простых способов визуализации иерархии заключается в выводе цепочки пар «подчиненный – его начальник», начиная с некоторого указанного работника и заканчивая руководителем организации. Это первый вариант такой функции. Его особенностью является тип возвращаемого значения, а именно: TABLE. При этом в заголовке функции указываются и атрибуты (поля) этой таблицы. Их всего два: emp_nbr и boss_emp_nbr, оба имеют тип INTEGER. Таким образом, при выводе цепочки пар «подчиненный – его начальник» мы получим только их числовые идентификаторы, а не имена.

```
-- ------------------------------------------------------------
-- Функция для обхода дерева снизу вверх, начиная с конкретного
-- узла.
-- ВАРИАНТ 1.
-- ------------------------------------------------------------
CREATE OR REPLACE FUNCTION up_tree_traversal(
  IN current_emp_nbr INTEGER )
  RETURNS TABLE( emp_nbr INTEGER, boss_emp_nbr INTEGER ) AS
$$
BEGIN
  -- Выбираем запись для текущего работника. На первой
  -- итерации это будет работник, с которого начинается
  -- обход дерева вверх.
  WHILE EXISTS ( SELECT *
                 FROM Org_chart AS O
                 WHERE O.emp_nbr = current_emp_nbr )
  LOOP
    -- Если нужно, то выполним какое-либо действие
    -- для текущего узла дерева (т. е. для текущего
    -- работника). Для этого нужно процедуру SomeProc
    -- заменить на какую-то полезную процедуру.
    -- CALL SomeProc (current_emp_nbr);

    -- Добавим очередную пару (работник; начальник)
    -- к формируемому множеству таких пар.
    -- ПРИМЕЧАНИЕ. Этот оператор RETURN не завершает
    -- выполнение процедуры, а лишь ДОБАВЛЯЕТ очередную
    -- запись (строку) к результирующей таблице.
    RETURN QUERY SELECT O.emp_nbr, O.boss_emp_nbr
                 FROM Org_chart AS O
                 WHERE O.emp_nbr = current_emp_nbr;
```

```
    -- Идем вверх по дереву к корню. Теперь текущим
    -- работником становится начальник только что
    -- обработанного работника, тем самым мы перемещаемся
    -- на один уровень вверх по дереву. Когда текущим
    -- работником станет главный начальник, у которого
    -- уже нет начальника, тогда результатом этого запроса
    -- будет current_emp_nbr = NULL, в результате чего
    -- условие цикла будет не выполнено, и цикл завершится.
    current_emp_nbr = ( SELECT O.boss_emp_nbr
                        FROM Org_chart AS O
                        WHERE O.emp_nbr = current_emp_nbr );
  END LOOP;
END;
$$
LANGUAGE plpgsql;
```

Это второй вариант предыдущей функции. Эта функция возвращает SETOF RECORD, т. е. множество записей. Других отличий от первого варианта нет.

```
-- ------------------------------------------------------------
-- Функция для обхода дерева снизу вверх, начиная с конкретного
-- узла.
-- ВАРИАНТ 2.
-- ------------------------------------------------------------
CREATE OR REPLACE FUNCTION up_tree_traversal2(
  IN current_emp_nbr INTEGER )
  RETURNS SETOF RECORD AS
$$
DECLARE
  -- В соответствии с выбранным типом результирующего значения,
  -- возвращаемого функцией, объявим переменную типа RECORD.
  rec RECORD;
BEGIN
  -- Выбираем запись для текущего работника. На первой
  -- итерации это будет работник, с которого начинается
  -- обход дерева вверх.
  WHILE EXISTS ( SELECT *
                 FROM Org_chart AS O
                 WHERE O.emp_nbr = current_emp_nbr )
  LOOP
    -- Если нужно, то выполним какое-либо действие
    -- для текущего узла дерева (т. е. для текущего
    -- работника). Для этого нужно процедуру SomeProc
    -- заменить на какую-то полезную процедуру.
    -- CALL SomeProc (current_emp_nbr);

    -- Добавим очередную пару (работник; начальник)
    -- к формируемому множеству таких пар.
```

```
    SELECT O.emp_nbr, O.boss_emp_nbr
    INTO rec
    FROM Org_chart AS O
    WHERE O.emp_nbr = current_emp_nbr;

    -- ПРИМЕЧАНИЕ. Этот оператор RETURN не завершает
    -- выполнение процедуры, а лишь ДОБАВЛЯЕТ очередную
    -- запись к результирующему множеству.
    RETURN NEXT rec;

    -- Идем вверх по дереву к корню. Теперь текущим
    -- работником становится начальник только что
    -- обработанного работника, тем самым мы перемещаемся
    -- на один уровень вверх по дереву. Когда текущим
    -- работником станет главный начальник, у которого
    -- уже нет начальника, тогда результатом этого запроса
    -- будет current_emp_nbr = NULL, в результате чего
    -- условие цикла будет не выполнено, и цикл завершится.
    current_emp_nbr = ( SELECT O.boss_emp_nbr
                        FROM Org_chart AS O
                        WHERE O.emp_nbr = current_emp_nbr );
  END LOOP;
  RETURN;
END;
$$
LANGUAGE plpgsql;
```

Одной из операций над иерархиями является удаление поддерева. Выполнение этой операции вручную, т. е. путем выполнения элементарных SQL-команд, может быть очень трудоемким делом, сопряженным с риском допустить ошибку. Функция, выполняющая эту операцию, не возвращает никакого значения, поэтому тип VOID. Важной особенностью этой функции является использование возможности отложить проверку выполнения ограничений FOREIGN KEY (внешних ключей) до конца выполнения функции. Поэтому эти внешние ключи и были объявлены с опцией DEFERRABLE в таблице Org_chart.

```
-- ------------------------------------------------------------
-- Функция для удаления поддерева.
-- ------------------------------------------------------------
CREATE OR REPLACE FUNCTION delete_subtree(
  IN dead_guy INTEGER ) RETURNS VOID AS
$$
-- Параметр dead_guy -- код работника, возглавляющего поддерево.
BEGIN
  -- Создадим врЕменную последовательность. Она нужна
  -- для того, чтобы формировать отрицательные значения
  -- для полей emp_nbr и boss_emp_nbr.
```

```
-- У J. Celko использовалось значение -99999, которое
-- записывалось в поля emp_nbr и boss_emp_nbr удаляемых
-- записей. Это значение служило меткой удаляемой записи.
-- Но мы добавили ограничение UNIQUE ( emp_nbr )
-- в таблицу Org_chart. Поэтому теперь уже стало
-- невозможно иметь более одной записи со значением поля
-- emp_nbr равным -99999. Мы вынуждены записывать в это поле
-- РАЗЛИЧНЫЕ значения в разных записях таблицы Org_chart.
-- А такие значения удобно формировать, используя
-- последовательность.
CREATE TEMP SEQUENCE New_emp_nbr START WITH 1;

-- Создадим временную таблицу.
CREATE TEMP TABLE Working_table ( emp_nbr INTEGER NOT NULL )
  ON COMMIT DROP;

-- Отложим проверку всех ограничений FOREIGN KEY
-- до конца транзакции, иначе СУБД не позволит нам
-- выполнять обновления полей emp_nbr и boss_emp_nbr:
-- мы будем записывать в них отрицательные значения,
-- а этих значений нет в таблице Personnel, на которую
-- ссылается таблица Org_chart.
SET CONSTRAINTS org_chart_emp_nbr_fkey,
                org_chart_boss_emp_nbr_fkey,
                org_chart_boss_emp_nbr_fkey1
  DEFERRED;

-- Пометим корень удаляемого поддерева и всех
-- непосредственных подчиненных путем записи в поле
-- emp_nbr или в поле boss_emp_nbr отрицательного значения,
-- формируемого с помощью последовательности.
UPDATE Org_chart
SET emp_nbr = CASE WHEN emp_nbr = dead_guy
                   THEN nextval( 'New_emp_nbr' ) * -1
                   ELSE emp_nbr
              END,
    boss_emp_nbr = CASE WHEN boss_emp_nbr = dead_guy
                   THEN nextval( 'New_emp_nbr' ) * -1
                   ELSE boss_emp_nbr
              END
-- Условие WHERE означает, что выбираются лишь записи,
-- в которых либо в поле emp_nbr, либо в поле boss_emp_nbr
-- стоит значение идентификатора "главы" удаляемого поддерева.
WHERE dead_guy IN ( emp_nbr, boss_emp_nbr );

-- Помечаем листья дерева, т. е. записи для работников,
-- не являющихся начальниками для других работников.
WHILE EXISTS ( SELECT * FROM Org_chart
               WHERE boss_emp_nbr < 0 AND emp_nbr >= 0 )
LOOP
  -- Получим список подчиненных следующего уровня.
```

```
    -- Сначала удалим все записи из временной таблицы.
    DELETE FROM Working_table;

    -- Выбираем подчиненных.
    INSERT INTO Working_table
      SELECT emp_nbr FROM Org_chart
      WHERE boss_emp_nbr < 0;

    -- Пометим следующий уровень подчиненных.
    -- Получаем очередное число из последовательности
    -- и умножаем его на -1, поскольку нам нужны
    -- отрицательные коды работников, т. к. именно
    -- отрицательные значения являются меткой удаляемой записи.
    UPDATE Org_chart
    SET emp_nbr = nextval( 'New_emp_nbr' ) * -1
    WHERE emp_nbr IN ( SELECT emp_nbr FROM Working_table );

    -- Помечаем начальников следующего уровня
    -- (при движении вниз по дереву).
    UPDATE Org_chart
    SET boss_emp_nbr = nextval( 'New_emp_nbr' ) * -1
    WHERE boss_emp_nbr IN ( SELECT emp_nbr FROM Working_table
);
  END LOOP;

  -- Удаляем все помеченные узлы.
  DELETE FROM Org_chart WHERE emp_nbr < 0;

  -- Снова активизируем все ограничения.
  SET CONSTRAINTS ALL IMMEDIATE;

  -- Удалим врЕменную последовательность.
  DROP SEQUENCE New_emp_nbr;
END;
$$
LANGUAGE plpgsql;
```

Использование представлений (views) позволяет упростить часто выполняемые запросы к базе данных. Приведенное далее представление выводит коды и имена работников в паре с соответствующим кодом и именем начальника.

```
-- ------------------------------------------------------------
-- Представление (VIEW) для реконструирования организационной
-- структуры.
-- ------------------------------------------------------------
DROP VIEW IF EXISTS Personnel_org_chart CASCADE;

CREATE VIEW Personnel_org_chart
 ( emp_nbr, emp, boss_emp_nbr, boss ) AS
```

```
  -- За основу принимается таблица Org_chart).
  -- ПРИМЕЧАНИЕ. LEFT OUTER JOIN необходим,
  -- т. к. у руководителя организации нет начальника
  -- и значение поля boss_emp_nbr у него NULL.
  SELECT O1.emp_nbr, E1.emp_name, O1.boss_emp_nbr, B1.emp_name
  FROM ( Org_chart AS O1 LEFT OUTER JOIN Personnel AS B1
         ON O1.boss_emp_nbr = B1.emp_nbr ), Personnel AS E1
  WHERE O1.emp_nbr = E1.emp_nbr;
```

А это представление предназначено для формирования всех возможных путей от вершины иерархии, т. е. от руководителя организации, до самого основания иерархии, т. е. до работников, не являющихся руководителями для других работников. Эта реализация представления имеет существенное ограничение: оно позволяет работать только с иерархиями, имеющими не более четырех уровней.

```
-- ------------------------------------------------------
-- Построение всех путей сверху дерева вниз
-- (только для четырех уровней иерархии)
-- ------------------------------------------------------
DROP VIEW IF EXISTS Create_paths;

CREATE VIEW Create_paths ( level1, level2, level3, level4 ) AS
  SELECT O1.emp AS e1, O2.emp AS e2, O3.emp AS e3,
         O4.emp AS e4
  FROM Personnel_org_chart AS O1
  LEFT OUTER JOIN Personnel_org_chart AS O2
    ON O1.emp = O2.boss
  LEFT OUTER JOIN Personnel_org_chart AS O3
    ON O2.emp = O3.boss
  LEFT OUTER JOIN Personnel_org_chart AS O4
    ON O3.emp = O4.boss
  -- Если закомментировать условие WHERE, тогда будут
  -- построены цепочки, начинающиеся с каждого работника,
  -- а не только с главного руководителя.
  WHERE O1.emp = 'Иван';
```

При формировании и модифицировании иерархий бывает необходимо перевести то или иное поддерево на более высокий уровень иерархии, т. е. к начальнику того работника, который возглавляет поддерево в настоящее время.

```
-- ------------------------------------------------------
-- Функция для удаления элемента иерархии и продвижения
-- дочерних элементов на один уровень вверх (т. е. к "бабушке").
-- ------------------------------------------------------
CREATE OR REPLACE FUNCTION delete_and_promote_subtree(
  IN dead_guy INTEGER ) RETURNS VOID AS
$$
```

```
-- Параметр dead_guy -- код работника, возглавляющего поддерево.
BEGIN
  -- Назначить нового начальника всем непосредственным
  -- подчиненным удаляемого работника.
  UPDATE Org_chart
  -- Получим код начальника для удаляемого работника.
  SET boss_emp_nbr = ( SELECT boss_emp_nbr
                       FROM Org_chart
                       WHERE emp_nbr = dead_guy
                     )
  WHERE boss_emp_nbr = dead_guy;

  -- Теперь удаляем работника. Все его подчиненные уже
  -- переподчинены вышестоящему начальнику.
  DELETE FROM Org_chart WHERE emp_nbr = dead_guy;
END;
$$
LANGUAGE plpgsql;
```

Контрольные вопросы и задания

1. Вызов функции можно использовать в подзапросе в предложении FROM. Напишите такой запрос с подзапросом на примере функции generate_students_data().

2. Какая команда используется для удаления функции? Удалите какую-нибудь вашу функцию.

3. Для чего нужны модификаторы (ключевые слова) IN и OUT перед именами параметров функций?

4. С помощью интерактивного терминала psql посмотрите список всех функций, созданных в вашей базе данных.

5. Каким образом можно сделать так, чтобы функция возвратила табличное значение?

6. Модифицируйте функцию count_letters(), подсчитывающую количество фамилий в таблице students («Студенты»), начинающихся на каждую букву. Сделайте так, чтобы в случае отсутствия в таблице фамилий, начинающихся с каких-то букв, в выводе функции эти буквы были представлены нулевыми (пустыми) значениями.

7. Что такое триггер?

8. Какие особенности имеет триггерная функция?

9. Чем триггеры уровня строки (row-level) отличаются от триггеров уровня команды (statement-level)?

10. Напишите триггер уровня строки (row-level) для таблицы «Студенты» или таблицы «Успеваемость».

11. В базе данных ais создайте таблицы, функции и триггеры, необходимые для изучения метода хранения иерархий в реляционных базах данных. Для этого можно поступить следующим образом. Сначала создайте текстовый файл с именем, например, adj_list.sql, содержащий все команды и определения функций, приведенные в тексте пособия, а затем выполните такую команду в среде операционной системы:

```
psql -d ais -f adj_list.sql
```

12. Сделайте выборки данных из таблиц «Персонал» и «Организационная структура», а также реконструируйте организационную структуру с помощью двух представлений (view). Команды можно выполнять не только в среде интерактивного терминала psql, но также и из командной строки операционной системы. Выполните эти команды в командной строке операционной системы:

```
psql -d ais -c "SELECT * FROM Personnel"
psql -d ais -c "SELECT * FROM Org_chart"
psql -d ais -c "SELECT * FROM Personnel_org_chart"
psql -d ais -c "SELECT * FROM Create_paths"
```

Не забудьте, что если не указан параметр -U, то утилита psql подключается к базе данных от имени пользователя базы данных, имя которого совпадает с именем пользователя операционной системы. Поэтому возможно, что вам придется использовать параметр -U, если в базе данных не создана учетная запись такого пользователя.

13. Выполните проверку структуры дерева на предмет отсутствия циклов с помощью функции tree_test().

```
SELECT * FROM tree_test();
```

Если вы еще не вносили изменения в таблицу «Организационная структура», то функция покажет отсутствие нарушения структуры дерева. Теперь создайте в таблице «Организационная структура» сначала короткий цикл, а затем длинный цикл. Для каждого из указанных циклов выполните проверку с помощью функции tree_test().

14. Выполните обход дерева организационной структуры снизу вверх, начиная с конкретного узла, можно с помощью функции up_tree_traversal() либо функции up_tree_traversal2(). Сначала сделайте это с помощью первой из функций:

```
SELECT * FROM up_tree_traversal( 6 );
```

Параметром этих функций является код работника. Измените код работника и повторите команду.

Теперь воспользуйтесь второй функцией. Учтите, что она возвращает SETOF RECORD, поэтому команда будет более сложной:

```
SELECT * FROM up_tree_traversal2( 6 ) AS (emp int, boss int);
```

Очевидно, что для использования числового кода работника нужно знать этот код. Удобнее иметь дело с именем работника. Поэтому можно в качестве параметра этих функций использовать подзапрос, возвращающий код работника в качестве своего результата. Не забудьте, что текст подзапроса заключается в скобки, поэтому появляются двойные скобки:

```
SELECT * FROM up_tree_traversal( ( SELECT … FROM Personnel
WHERE …) );
```

Завершите эту команду и выполните ее с различными именами работников.

15. Выполните операцию удаления поддерева с помощью функции delete_subtree(). Параметром функции является код работника.

```
SELECT * FROM delete_subtree( 6 );
```

Аналогично работе с функцией up_tree_traversal() используйте подзапрос для получения кода работника по его имени. После удаления поддерева посмотрите, что стало с организационной структурой, с помощью двух представлений Personnel_org_chart и Create_paths.

16. Если в таблице «Организационная структура» осталось мало данных, то дополните ее данными и выполните удаление элемента иерархии и продвижение дочерних элементов на один уровень вверх (т. е. к «бабушке»).

```
SELECT * FROM delete_and_promote_subtree( 5 );
```

Аналогично работе с функцией up_tree_traversal() используйте подзапрос для получения кода работника по его имени.

После удаления элемента иерархии посмотрите, что стало с организационной структурой, с помощью двух представлений Personnel_org_chart и Create_paths.

17. Представление Create_paths позволяет отобразить только четыре уровня иерархии. Модифицируйте его так, чтобы оно могло работать с пятью уровнями иерархии.

18. Самостоятельно ознакомьтесь с таким средством работы с таблицами базы данных, как курсоры (cursors). Воспользуйтесь технической документацией на PostgreSQL, глава «PL/pgSQL – SQL Procedural Language». Напишите небольшую функцию с применением курсора.

19. Самостоятельно ознакомьтесь с таким средством работы с таблицами базы данных, как правила (rules). Воспользуйтесь технической документацией на PostgreSQL, глава «The Rule System». Напишите правила, позволяющие организовать журнальные таблицы в базе данных «Студенты».

ЗАКЛЮЧЕНИЕ

В учебном пособии были рассмотрены многие важные вопросы и технологии, которыми должен владеть администратор информационной системы. Однако это пособие призвано дать лишь начальные знания в этой сфере. Поэтому после изучения представленного учебного материала имеет смысл продолжить изучение более сложных вопросов самостоятельно с использованием технической документации на СУБД PostgreSQL. Мы можем лишь порекомендовать те вопросы, которые вы могли бы рассмотреть.

В современных условиях в базах данных зачастую необходимо хранить данные, представленные на различных языках народов мира. Поэтому важным вопросом является обеспечение корректной локализации сервера и клиентских приложений.

С ростом объемов данных и повышением требований к их сохранности и доступности для приложений все более актуальными становятся такие технологии, как обеспечение высокой доступности (High Availability), балансировка нагрузки (Load Balancing) и репликация данных (Replication). Важно также обеспечить эффективное выполнение процедур резервного копирования, выполняемых на базах данных большого объема.

Очень большое значение для эффективного выполнения функций администратора имеет знание и понимание внутреннего устройства сервера баз данных. Администратор должен хорошо ориентироваться в многочисленных системных таблицах, содержащих важную информацию о текущем состоянии кластера баз данных. И наконец, без уверенного владения языком SQL выполнение административных функций вряд ли возможно.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Беленькая, М. Н. Администрирование в информационных системах [Текст] : учеб. пособие для вузов / М. Н. Беленькая, С. Т. Малиновский, Н. В. Яковенко. – М. : Горячая линия – Телеком, 2011. – 400 с. : ил.

2. Ригс, С. Администрирование PostgreSQL 9. Книга рецептов : [Текст] : пер. с англ. / С. Ригс, Х. Кроссинг. – М. : ДМК Пресс, 2012. – 368 с. : ил.

3. Селко, Д. Стиль программирования Джо Селко на SQL [Текст] : пер. с англ. / Д. Селко. – М. : Русская редакция ; СПб. : Питер, 2006. – 206 с. : ил.

4. Уорсли, Дж. PostrgeSQL. Для профессионалов [Текст] : пер. с англ. / Дж. Уорсли, Дж. Дрейк. – СПб. : Питер, 2003. – 496 с. : ил.

5. Celko, J. Joe Celko's Trees and hierarchies in SQL for smarties [Text] / Joe Celko. – San Francisco : Morgan Kaufmann, 2004. – 225 p. – il.

6. PostgreSQL [Электронный ресурс] : официальный сайт / The PostgreSQL Global Development Group. – Электрон. дан. – Б. м., [1996–]. – Режим доступа: http://www.postgresql.org. – Загл. с экрана.