IV Пробуем SQL

Подключение с помощью psql

Чтобы подключиться к серверу СУБД и выполнить какие-либо команды, требуется программа-клиент. В главе «PostgreSQL для приложения» мы будем говорить о том, как посылать запросы из программ на разных языках программирования, а сейчас речь пойдет о терминальном клиенте `psql`, работа с которым происходит интерактивно в режиме командной строки.

К сожалению, в наше время многие недолюбливают командную строку. Почему имеет смысл научиться с ней работать?

Во-первых, `psql` — стандартный клиент, он входит в любую сборку PostgreSQL и поэтому всегда под рукой. Иметь настроенную под себя среду — это, конечно, хорошо, но оказаться беспомощным в среде незнакомой просто нелогично.

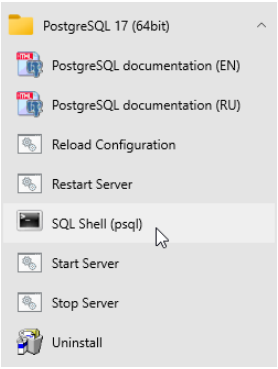
Во-вторых, `psql` действительно удобен для решения повседневных задач по администрированию баз данных, для написания небольших запросов и для автоматизации процессов, например, периодической установки обновлений программного кода на сервер СУБД. Он имеет собственные команды, позволяющие сориентироваться в объектах,

хранящихся в базе данных, и представить информацию из таблиц в наглядном виде.

Но если вы привыкли работать с графическими пользовательскими интерфейсами, попробуйте pgAdmin — мы еще упомянем эту программу ниже — или другие аналогичные продукты: wiki.postgresql.org/wiki/Community_Guide_to_PostgreSQL_GUI_Tools

Чтобы запустить `psql` в операционной системе Linux, выполните команду:

```
$ sudo -u postgres psql
```



В ОС Windows запустите программу «SQL Shell (psql)» из меню «Пуск». В ответ на запрос введите пароль пользователя `postgres`, указанный при установке PostgreSQL.

Пользователи Windows могут столкнуться с проблемой неправильного отображения символов кириллицы в терминале. В этом случае убедитесь, что свойствах окна терминала установлен TrueType-шрифт (обычно «Lucida Console» или «Consolas»).

Итак, приглашение выглядит одинаково в обеих операционных системах: `postgres=#` («postgres» здесь — это имя

базы данных, к которой вы сейчас подключены). Один сервер может обслуживать несколько БД, но работать в каждый момент времени можно только с одной.

Теперь изучим первые команды. Вводите только то, что выделено жирным; приглашение и ответ системы на команду приведены исключительно для удобства.

База данных

Создадим новую базу данных с именем test. Выполните:

```
postgres=# CREATE DATABASE test;
CREATE DATABASE
```

Не забудьте про точку с запятой в конце команды — пока PostgreSQL не увидит этот символ, он будет считать, что вы продолжаете ввод (то есть команда может быть разбита на несколько строк).

Теперь переключимся на созданную базу:

```
postgres=# \c test
You are now connected to database "test" as user
"postgres".
test=#
```

Как видите, приглашение сменилось на `test=#`.

Команда, которую мы только что ввели, не похожа на SQL — она начинается с обратной косой черты. Так выглядят специальные команды, которые понимает только `psql` (поэтому, если у вас открыт pgAdmin или другое графическое средство, пропускайте все, что начинается с косой черты, или поищите замену).

Команд `psql` довольно много; с некоторыми из них мы познакомимся чуть позже, а полный список с краткими описаниями можно получить прямо сейчас:

```
test=# \?
```

Поскольку справочная информация довольна объемна, она будет показана с помощью настроенной в операционной системе команды-пейджера (обычно `more` или `less`).

Таблицы

В реляционных СУБД данные представляются в виде **таблиц**. Структура таблицы определяется ее **столбцами**. Собственно данные располагаются в **строках;** они хранятся неупорядоченными и даже не обязательно располагаются в порядке их добавления в таблицу.

Для каждого столбца устанавливается **тип данных**; значения полей в строках должны соответствовать этим типам. PostgreSQL располагает большим числом встроенных типов (postgrespro.ru/doc/datatype) и возможностями для создания новых, но мы ограничимся самыми основными:

- integer — целые числа;
- text — текстовые строки;
- boolean — логический тип, принимающий значения `true` («истинно») или `false` («ложно»).

Помимо обычных значений, определяемых типом данных, поле может иметь **неопределенное значение** NULL — его можно рассматривать как «значение неизвестно» или «значение не задано».

Давайте создадим таблицу дисциплин, читаемых в вузе:

```
test=# CREATE TABLE courses(
test(#   c_no text PRIMARY KEY,
test(#   title text,
test(#   hours integer
test(# );
CREATE TABLE
```

Обратите внимание, как меняется приглашение `psql`: это подсказка, что ввод команды продолжается на новой строке. В дальнейшем для удобства мы не будем дублировать приглашение на каждой строке.

Этой командой мы определили, что таблица с именем `courses` будет состоять из трех столбцов: `c_no` — текстовый номер курса, `title` — название курса, и `hours` — целое число лекционных часов.

Кроме столбцов и типов данных, можно ввести ограничения целостности, которые будут проверяться автоматически, — СУБД не допустит появления в базе некорректных данных. В нашем примере добавлено ограничение `PRIMARY KEY` для столбца `c_no`; теперь в нем не допускаются повторяющиеся, а также неопределенные значения. С помощью такого столбца можно отличать строки друг от друга. Полный список ограничений целостности есть на странице postgrespro.ru/doc/ddl-constraints.

Точный синтаксис команды `CREATE TABLE` можно посмотреть в документации, а можно прямо в `psql`:

```
test=# \help CREATE TABLE
```

Такая справка есть по каждой команде SQL, а полный список команд покажет `\help` без параметров.

Наполнение таблиц

Добавим в созданную таблицу несколько строк:

```
test=# INSERT INTO courses(c_no, title, hours)
VALUES ('CS301', 'Базы данных', 30),
       ('CS305', 'Сети ЭВМ', 60);
```

```
INSERT 0 2
```

Для массовой загрузки данных из внешнего источника команда INSERT подходит плохо, зато есть специально предназначенная для этого команда COPY: postgrespro.ru/doc/sql-copy.

Создадим в базе еще две таблицы: «Студенты» и «Экзамены». Пусть по каждому студенту хранится его имя и год поступления, а идентифицироваться он будет номером студенческого билета.

```
test=# CREATE TABLE students(
  s_id integer PRIMARY KEY,
  name text,
  start_year integer
);
```

```
CREATE TABLE
```

```
test=# INSERT INTO students(s_id, name, start_year)
VALUES (1451, 'Анна', 2014),
       (1432, 'Виктор', 2014),
       (1556, 'Нина', 2015);
```

```
INSERT 0 3
```

Таблица экзаменов содержит данные об оценках, полученных студентами по различным дисциплинам. Таким образом, студенты и дисциплины связаны друг с другом отношением «многие ко многим»: один студент может сдавать

экзамены по многим дисциплинам, а экзамен по одной дисциплине могут сдавать много студентов.

Запись в таблице экзаменов идентифицируется совокупностью номера студбилета и номера курса. Такое ограничение целостности, относящее сразу к нескольким столбцам, определяется с помощью предложения CONSTRAINT:

```
test=# CREATE TABLE exams(
  s_id integer REFERENCES students(s_id),
  c_no text REFERENCES courses(c_no),
  score integer,
  CONSTRAINT pk PRIMARY KEY(s_id, c_no)
);
CREATE TABLE
```

Кроме того, с помощью предложения REFERENCES мы добавили два ограничения ссылочной целостности, называемые **внешними ключами**. Такие ограничения показывают, что значения в одной таблице **ссылаются** на строки в другой таблице.

Теперь при любых действиях СУБД будет проверять соответствие всех идентификаторов s_id, указанных в таблице экзаменов, реальным студентам (то есть записям в таблице студентов), а также номера c_no — реальным курсам. Таким образом, будет исключена возможность поставить оценку несуществующему студенту или же по несуществующей дисциплине — независимо от действий пользователя или возможных ошибок в приложении.

Поставим нашим студентам несколько оценок:

```
test=# INSERT INTO exams(s_id, c_no, score)
VALUES (1451, 'CS301', 5),
       (1556, 'CS301', 5),
       (1451, 'CS305', 5),
       (1432, 'CS305', 4);
INSERT 0 4
```

Выборка данных

Простые запросы

Чтение данных из таблиц выполняется оператором SQL SELECT. Для примера выведем только два столбца из таблицы courses. Конструкция AS позволяет переименовать столбец, если это необходимо:

```
test=# SELECT title AS course_title, hours
FROM courses;
 course_title | hours
--------------+-------
 Базы данных  |    30
 Сети ЭВМ     |    60
(2 rows)
```

Чтобы вывести все столбцы, достаточно указать символ звездочки:

```
test=# SELECT * FROM courses;
 c_no  |    title    | hours
-------+-------------+-------
 CS301 | Базы данных |    30
 CS305 | Сети ЭВМ    |    60
(2 rows)
```

В промышленном коде лучше явно перечислять только необходимые столбцы, чтобы запрос выполнялся эффективнее, а результат не зависел от появления новых столбцов. Но для интерактивных запросов «звездочка» очень удобна.

Выдача по запросу может содержать одинаковые строки. Когда выводятся не все столбцы — дубликаты могут появиться даже если в исходной таблице их не было:

```
test=# SELECT start_year FROM students;
 start_year
------------
       2014
       2014
       2015
(3 rows)
```

Чтобы выбрать все **различные** года поступления, после SELECT надо добавить слово DISTINCT:

```
test=# SELECT DISTINCT start_year FROM students;
 start_year
------------
       2014
       2015
(2 rows)
```

Подробнее смотрите в документации: postgrespro.ru/doc/sql-select#SQL-DISTINCT

Вообще после слова SELECT можно указывать любые выражения. А без предложения FROM запрос вернет одну строку. Например:

```
test=# SELECT 2+2 AS result;
 result
--------
      4
(1 row)
```

Обычно при выборке данных требуется получить не все строки, а только те, которые удовлетворят какому-либо условию. Такое условие фильтрации записывается в предложении WHERE:

```
test=# SELECT * FROM courses WHERE hours > 45;
```

```
 c_no  |  title   | hours
-------+----------+-------
 CS305 | Сети ЭВМ |    60
(1 row)
```

Условие должно иметь логический тип. Например, оно может содержать операторы =, <> (или !=), >, >=, <, <=, а также может объединять более простые условия с помощью логических операций AND, OR, NOT и круглых скобок — как в обычных языках программирования.

Тонкий момент представляет собой неопределенное значение NULL. В выборку попадают только те строки, для которых условие фильтрации истинно; если же значение ложно **или не определено,** строка отбрасывается.

Учтите:

- результат сравнения чего-либо с неопределенным значением не определен;
- результат логических операций с неопределенным значением, как правило, не определен (исключения: `true OR NULL = true`, `false AND NULL = false`);
- для проверки определенности значения используются специальные операторы `IS NULL` (`IS NOT NULL`) и `IS DISTINCT FROM` (`IS NOT DISTINCT FROM`).

При работе с неопределенными значениями часто используют выражение `coalesce` (читается «коуэлес») для замены NULL на что-нибудь другое, например, на пустую строку для текстовых типов или на ноль для числовых.

Подробнее смотрите в документации: postgrespro.ru/doc/functions-comparison.

Соединения

Грамотно спроектированная реляционная база данных не содержит избыточной информации. Например, таблица экзаменов не должна содержать имя студента, потому что его можно найти в другой таблице по номеру студенческого билета.

Поэтому для получения всех необходимых значений в запросе часто приходится соединять данные из нескольких таблиц, перечисляя их имена в предложении FROM:

```
test=# SELECT * FROM courses, exams;
 c_no  |    title    | hours | s_id | c_no  | score
-------+-------------+-------+------+-------+-------
 CS301 | Базы данных |    30 | 1451 | CS301 |     5
 CS305 | Сети ЭВМ    |    60 | 1451 | CS301 |     5
 CS301 | Базы данных |    30 | 1556 | CS301 |     5
 CS305 | Сети ЭВМ    |    60 | 1556 | CS301 |     5
 CS301 | Базы данных |    30 | 1451 | CS305 |     5
 CS305 | Сети ЭВМ    |    60 | 1451 | CS305 |     5
 CS301 | Базы данных |    30 | 1432 | CS305 |     4
 CS305 | Сети ЭВМ    |    60 | 1432 | CS305 |     4
(8 rows)
```

То, что у нас получилось, называется прямым или декартовым произведением таблиц — к каждой строке одной таблицы добавляется каждая строка другой.

Как правило, более полезный и содержательный результат можно получить, указав в предложении WHERE условие соединения. Запросим оценки по всем дисциплинам, сопоставляя курсы с теми экзаменами, которые проводились именно по данному курсу:

```
test=# SELECT courses.title, exams.s_id, exams.score
FROM courses, exams
WHERE courses.c_no = exams.c_no;
```

```
    title    | s_id | score
-------------+------+-------
 Базы данных | 1451 |     5
 Базы данных | 1556 |     5
 Сети ЭВМ    | 1451 |     5
 Сети ЭВМ    | 1432 |     4
(4 rows)
```

Запросы можно формулировать и в другом виде, указывая соединения с помощью ключевого слова JOIN. Выведем студентов и их оценки по курсу «Сети ЭВМ»:

```
test=# SELECT students.name, exams.score
FROM students
JOIN exams
  ON students.s_id = exams.s_id
  AND exams.c_no = 'CS305';
  name  | score
--------+-------
 Анна   |     5
 Виктор |     4
(2 rows)
```

С точки зрения СУБД эти формы эквивалентны друг другу, так что можно использовать тот способ, который представляется более наглядным.

Как видно, в выдаче нет тех строк таблицы, указанной слева от слова JOIN, для которых не нашлось пары в таблице, указанной справа от этого слова: условие наложено на дисциплины, но исключаются и студенты, не сдававшие по ней экзамен. Чтобы в выборку попали все студенты, надо использовать внешнее соединение:

```
test=# SELECT students.name, exams.score
FROM students
LEFT JOIN exams
  ON students.s_id = exams.s_id
  AND exams.c_no = 'CS305';
```

```
  name   | score
---------+-------
 Анна    |     5
 Виктор  |     4
 Нина    |
(3 rows)
```

Теперь в выдаче есть и те строки из левой таблицы (поэтому операция называется LEFT JOIN), для которых не нашлось пары в правой. При этом для столбцов правой таблицы возвращаются неопределенные значения.

Условия после WHERE применяются к уже соединенным строкам, поэтому, если вынести ограничение на дисциплины из условия соединения в предложение WHERE, Нина не попадет в выборку — ведь для нее значение exams.c_no не определено:

```
test=# SELECT students.name, exams.score
FROM students
LEFT JOIN exams ON students.s_id = exams.s_id
WHERE exams.c_no = 'CS305';

  name   | score
---------+-------
 Анна    |     5
 Виктор  |     4
(2 rows)
```

Не стоит опасаться соединений. Это обычная и естественная для реляционных СУБД операция, и у PostgreSQL имеется целый арсенал эффективных механизмов для ее выполнения. Не соединяйте данные в приложении, доверьте эту работу серверу баз данных — он прекрасно с ней справляется.

Подробнее смотрите в документации: postgrespro.ru/doc/sql-select#SQL-FROM.

Подзапросы

Оператор SELECT формирует таблицу, которая (как мы уже видели) может быть выведена в качестве результата, а может быть использована в другой конструкции языка SQL в любом месте, где по смыслу может находиться таблица. Такая вложенная команда SELECT, заключенная в круглые скобки, называется **подзапросом.**

Если подзапрос возвращает ровно одну строку и ровно один столбец, его можно использовать как обычное скалярное выражение:

```
test=# SELECT name,
  (SELECT score
   FROM exams
   WHERE exams.s_id = students.s_id
   AND exams.c_no = 'CS305')
FROM students;

  name   | score
---------+-------
 Анна    |     5
 Виктор  |     4
 Нина    |
(3 rows)
```

Если скалярный подзапрос, использованный в списке выражений SELECT, не содержит ни одной строки, возвращается неопределенное значение (как в последней строке выдачи по нашему примеру). Поэтому скалярные подзапросы можно раскрыть, заменив их на соединение, но обязательно внешнее.

Скалярные подзапросы можно также использовать в условиях фильтрации. Получим все экзамены, которые сдавали студенты, поступившие после 2014 года:

```
test=# SELECT *
FROM exams
WHERE (SELECT start_year
       FROM students
       WHERE students.s_id = exams.s_id) > 2014;
```

```
 s_id | c_no  | score
------+-------+-------
 1556 | CS301 |     5
(1 row)
```

В SQL можно формулировать условия и на подзапросы, возвращающие произвольное количество строк. Для этого существует несколько конструкций, одна из которых — отношение IN — проверяет, содержится ли значение в таблице, возвращаемой подзапросом.

Выведем студентов, получивших какие-либо оценки по указанному курсу:

```
test=# SELECT name, start_year
FROM students
WHERE s_id IN (SELECT s_id
               FROM exams
               WHERE c_no = 'CS305');
```

```
  name  | start_year
--------+------------
 Анна   |       2014
 Виктор |       2014
(2 rows)
```

Оператор NOT IN возвращает противоположный результат. Например, список студентов, не получивших ни одной отличной оценки:

```
test=# SELECT name, start_year
FROM students
WHERE s_id NOT IN
      (SELECT s_id FROM exams WHERE score = 5);
```

```
  name  | start_year
--------+------------
 Виктор |       2014
(1 row)
```

Обратите внимание, что в выборку попадут и студенты, не получившие ни одной оценки.

Еще одна возможность — использовать предикат EXISTS, проверяющий, возвратил ли подзапрос хотя бы одну строку. С его помощью можно записать предыдущий запрос в другом виде:

```
test=# SELECT name, start_year
FROM students
WHERE NOT EXISTS (SELECT s_id
                  FROM exams
                  WHERE exams.s_id = students.s_id
                  AND score = 5);
```

```
  name  | start_year
--------+------------
 Виктор |       2014
(1 row)
```

Подробнее смотрите в документации: postgrespro.ru/doc/functions-subquery.

Выше мы дополняли имена столбцов именами таблиц, чтобы избежать неоднозначности, но иногда этого недостаточно. Например, одна и та же таблица может фигурировать в запросе дважды, или на месте таблицы в предложении FROM может стоять безымянный подзапрос. В таких случаях после подзапроса можно указывать его псевдоним (alias). Обычным таблицам тоже можно присваивать псевдонимы.

Выведем имена студентов и их оценки по предмету «Базы данных»:

```
test=# SELECT s.name, ce.score
FROM students s
JOIN (SELECT exams.*
      FROM courses, exams
      WHERE courses.c_no = exams.c_no
      AND courses.title = 'Базы данных') ce
  ON s.s_id = ce.s_id;
 name | score
------+-------
 Анна |     5
 Нина |     5
(2 rows)
```

Здесь s — псевдоним таблицы, а ce — псевдоним подзапроса. Псевдонимы лучше делать короткими, но понятными.

Тот же запрос можно записать и без подзапросов:

```
test=# SELECT s.name, e.score
FROM students s, courses c, exams e
WHERE c.c_no = e.c_no
AND c.title = 'Базы данных'
AND s.s_id = e.s_id;
```

Сортировка

Как уже говорилось, данные в таблицах не упорядочены. Для вывода строк результата в определенном порядке используется предложение ORDER BY со списком выражений, по которым надо выполнить сортировку. После каждого выражения (ключа сортировки) можно указать направление: ASC — по возрастанию (этот порядок используется по умолчанию) — или DESC — по убыванию.

```
test=# SELECT *
FROM exams
ORDER BY score, s_id, c_no DESC;
```

```
 s_id | c_no  | score
------+-------+-------
 1432 | CS305 |     4
 1451 | CS305 |     5
 1451 | CS301 |     5
 1556 | CS301 |     5
(4 rows)
```

Здесь строки упорядочены по возрастанию оценки; если оценки совпадают — по возрастанию номера студенческого билета; если же совпадают оба первых ключа — по убыванию номера курса.

Операцию сортировки имеет смысл выполнять в конце запроса непосредственно перед получением результата; в подзапросах она обычно бессмысленна.

Подробнее смотрите в документации: postgrespro.ru/doc/sql-select#SQL-ORDERBY.

Группировка

При группировке в одной строке результата размещается значение, вычисленное на основании данных нескольких строк исходных таблиц. Вместе с группировкой используют **агрегатные функции**. Например, выведем общее количество проведенных экзаменов, количество сдававших их студентов и средний балл:

```
test=# SELECT count(*), count(DISTINCT s_id),
avg(score)
FROM exams;
 count | count |        avg
-------+-------+--------------------
     4 |     3 | 4.7500000000000000
(1 row)
```

Ту же информацию можно получить в разбивке по номерам курсов с помощью предложения GROUP BY, в котором указываются ключи группировки:

```
test=# SELECT c_no, count(*),
count(DISTINCT s_id), avg(score)
FROM exams
GROUP BY c_no;
```

```
 c_no  | count | count |        avg
-------+-------+-------+--------------------
 CS301 |     2 |     2 | 5.0000000000000000
 CS305 |     2 |     2 | 4.5000000000000000
(2 rows)
```

Полный список агрегатных функций: postgrespro.ru/doc/functions-aggregate.

При использовании группировки может возникнуть необходимость отфильтровать строки на основании результатов агрегирования. Такие условия можно задать в предложении HAVING. Отличие от WHERE состоит в том, что условия WHERE применяются до группировки (в них можно использовать столбцы исходных таблиц), а условия HAVING — после группировки (и в них также можно использовать столбцы таблицы-результата).

Выберем имена студентов, получивших более одной пятерки по любому предмету:

```
test=# SELECT students.name
FROM students, exams
WHERE students.s_id = exams.s_id AND exams.score = 5
GROUP BY students.name
HAVING count(*) > 1;
```

```
 name
------
 Анна
(1 row)
```

Подробнее смотрите в документации: postgrespro.ru/doc/sql-select#SQL-GROUPBY.

Изменение и удаление данных

Изменение данных в таблице выполняет оператор UPDATE, в котором указываются новые значения полей для строк, определяемых предложением WHERE (таким же, как в операторе SELECT).

Например, увеличим число лекционных часов для курса «Базы данных» в два раза:

```
test=# UPDATE courses
SET hours = hours * 2
WHERE c_no = 'CS301';
```

```
UPDATE 1
```

Подробнее смотрите в документации: postgrespro.ru/doc/sql-update.

Оператор DELETE удаляет из указанной таблицы строки, определяемые все тем же предложением WHERE:

```
test=# DELETE FROM exams WHERE score < 5;
```

```
DELETE 1
```

Транзакции

Теперь усложним схему данных и распределим студентов по группам, причем у каждой группы должен быть староста. Для этого создадим таблицу групп:

```
test=# CREATE TABLE groups(
  g_no text PRIMARY KEY,
  monitor integer NOT NULL REFERENCES students(s_id)
);
CREATE TABLE
```

Здесь мы использовали ограничение целостности NOT NULL, которое запрещает неопределенные значения.

Теперь нам нужен еще один столбец — номер группы. К счастью, в уже существующую таблицу можно добавить новый столбец:

```
test=# ALTER TABLE students
ADD g_no text REFERENCES groups(g_no);
ALTER TABLE
```

С помощью команды psql всегда можно посмотреть, какие столбцы определены в таблице:

```
test=# \d students
      Table "public.students"
   Column   |  Type   | Modifiers
------------+---------+----------
 s_id       | integer | not null
 name       | text    |
 start_year | integer |
 g_no       | text    |
 ...
```

Также можно вспомнить, какие вообще таблицы присутствуют в базе данных:

```
test=# \d
          List of relations
 Schema |   Name   | Type  | Owner
--------+----------+-------+----------
 public | courses  | table | postgres
```

```
 public | exams    | table | postgres
 public | groups   | table | postgres
 public | students | table | postgres
(4 rows)
```

Создадим теперь группу «А-101» и поместим в нее всех студентов, а старостой сделаем Анну.

Здесь есть нетривиальный момент. Нельзя создать группу, не указав старосты, но нельзя и назначить студента старостой группы, в которую он не входит, — это привело бы к появлению в базе данных логически некорректных, несогласованных данных. Эти две операции не имеют смысла по отдельности: их надо совершить одновременно. Группа операций, составляющих логически неделимую единицу работы, называется **транзакцией**.

Начнем транзакцию:

```
test=# BEGIN;

BEGIN
```

Затем добавим группу вместе со старостой. Помнить наизусть номера студенческих мы не обязаны, так что выполним запрос прямо в команде добавления строк:

```
test=*# INSERT INTO groups(g_no, monitor)
SELECT 'A-101', s_id
FROM students
WHERE name = 'Анна';

INSERT 0 1
```

«Звездочка» в приглашении напоминает о незавершенной транзакции.

Теперь откройте новое окно терминала и запустите еще один процесс `psql`: это будет сеанс, работающий параллельно с первым. Чтобы не запутаться, команды второго сеанса мы будем показывать с отступом.

Увидит ли второй сеанс сделанные изменения?

```
postgres=# \c test
You are now connected to database "test" as user
"postgres".
test=# SELECT * FROM groups;
 g_no | monitor
------+---------
(0 rows)
```

Нет, не увидит, ведь транзакция еще не завершена.

Теперь переведем всех студентов в созданную группу:

```
test=*# UPDATE students SET g_no = 'A-101';
UPDATE 3
```

И снова второй сеанс видит согласованные данные, актуальные на начало еще не оконченной транзакции:

```
test=# SELECT * FROM students;
 s_id |  name  | start_year | g_no
------+--------+------------+------
 1451 | Анна   |       2014 |
 1432 | Виктор |       2014 |
 1556 | Нина   |       2015 |
(3 rows)
```

А теперь завершим транзакцию, зафиксировав все сделанные изменения:

```
test=*# COMMIT;
COMMIT
```

И только в этот момент второму сеансу становятся доступны все изменения, сделанные в транзакции, как будто они появились одномоментно:

```
test=# SELECT * FROM groups;
  g_no | monitor
-------+---------
 A-101 |    1451
(1 row)

test=# SELECT * FROM students;
 s_id |  name  | start_year | g_no
------+--------+------------+-------
 1451 | Анна   |       2014 | A-101
 1432 | Виктор |       2014 | A-101
 1556 | Нина   |       2015 | A-101
(3 rows)
```

СУБД дает несколько очень важных гарантий.

Во-первых, любая транзакция либо выполняется целиком (как в нашем примере), либо не выполняется никак. Если бы при выполнении одной из команд произошла ошибка, или мы сами прервали бы транзакцию командой ROLLBACK, то база данных осталась бы в том состоянии, в котором она была до команды BEGIN. Это свойство называется **атомарностью**.

Во-вторых, при фиксации изменений должны выполняться все ограничения целостности, иначе транзакция обрывается. Как в начале работы транзакции, так и в конце ее данные обязательно находятся в согласованном состоянии; это свойство так и называется — **согласованность**.

В-третьих, как мы убедились на примере, другие пользователи никогда не увидят данные, еще не зафиксированные транзакцией. Это свойство называется **изоляцией**; за счет его соблюдения СУБД способна параллельно обслуживать много сеансов, не жертвуя корректностью данных. Особенностью PostgreSQL является очень эффективная реализация изоляции: несколько сеансов могут одновременно читать и изменять данные, не блокируя друг друга. Блокировка возникает только при попытке одновременного изменения одной и той же строки несколькими разными процессами.

И в-четвертых, гарантируется **долговечность**: зафиксированные данные не пропадут даже в случае сбоя (конечно, при правильных настройках и регулярном выполнении резервного копирования).

Это крайне полезные свойства, без которых невозможно представить себе реляционную систему баз данных.

Подробнее о транзакциях см. postgrespro.ru/doc/tutorial-transactions (и еще более подробно — postgrespro.ru/doc/mvcc).

Полезные команды psql

\?	Справка по командам `psql`.
\h	Справка по SQL: список доступных команд или синтаксис конкретной команды.
\x	Переключение табличного вывода (столбцы и строки) на расширенный (каждый столбец на отдельной строке) и обратно. Удобно для просмотра нескольких «широких» строк.

\l	Список баз данных.
\du	Список пользователей.
\dt	Список таблиц.
\di	Список индексов.
\dv	Список представлений.
\df	Список функций.
\dn	Список схем.
\dx	Список установленных расширений.
\dp	Список привилегий.
\d имя	Подробная информация по конкретному объекту базы данных.
\d+ имя	Еще более подробная информация по конкретному объекту.
\timing on	Вывод времени выполнения операторов.

Заключение

Конечно, мы успели осветить только малую толику того, что необходимо знать о СУБД, но надеемся, что вы убедились: начать использовать PostgreSQL совсем нетрудно. Язык SQL позволяет формулировать запросы самой разной сложности, а PostgreSQL предоставляет качественную поддержку стандарта и эффективную реализацию. Пробуйте, экспериментируйте!

И еще одна важная команда `psql`: чтобы завершить сеанс работы, наберите

```
test=# \q
```