Практическое задание 9

PostgreSQL

Разработка базы данных

Цель. Изучить способы создания таблиц, запросов и представлений в СУБД

Задачи:

- создать таблицы, заполнить их информацией,
- реализовать выборку (запрос) всех данных,
- реализовать запрос с условием,
- реализовать запрос с использованием псевдонимов,
- реализовать запрос с использованием подзапроса,
- реализовать запрос с использованием сортировки,
- реализовать запрос с использованием группировки,
- реализовать запрос с выбором неповторяющихся записей,
- реализовать запрос с использованием агрегатных функций, •реализовать транзакцию,
- создать представления.

Краткие теоретические сведения

Важное замечание - задания данной лабораторной работы, в полном объеме, выполняются в консоли SQL Shell (psql).

Как вы знаете, язык реляционных баз данных – SQL, состоит и нескольких разделов – самых известных (DDL, DML, DQL) и других, более утилитарных, для каждой конкретной СУБД. ***Раздел DDL (Data Definition Language)*** позволяет определять объекты (данные) в базе с помощью операторов CREATE, DROP, созданные объектами (данными) с помощью операторов INSERT, DELETE, UPDATE, а раздел ***DQL (Data Query Language)*** позволяет

реализовывать запросы к имеющимся в базе объектам и извлекать с помощью оператора SELECT данные для анализа.

В лабораторной работе вы достаточно подробно разберетесь в особенностях применения оператора запросов SELECT. С помощью этого оператора реализуются любые конструкции запросов. Большой блок справочной информации об операторе запросов вы найдете по адресу

Кроме того, вы познакомитесь с транзакциями — фундаментальным понятием для любых СУБД. Смысл транзакции в том, что она объединяет последовательность действий в одну операцию «всё или ничего». Промежуточные состояния внутри последовательности не видны другим транзакциям, а если что-то помешает успешно завершить транзакцию, ни один из результатов этих действий не сохранится в базе данных. Развернутую информацию об этом понятии, в применении к СУБД PostgreSQL, вы можете найти по адресу postgrespro.ru/docs/postgresql/15/tutorial-transactions. Также вы

Создание таблиц

Создайте таблицу дисциплин, читаемых в вузе

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

Обратите внимание, как меняется приглашение psql: это подсказка, что ввод команды продолжается на новой строке. В этой команде 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 мы можем определить ограничения целостности, которые будут автоматически проверяться — СУБД не допустит появление в базе некорректных данных. В нашем примере мы добавили ограничение первичного ключа PRIMARY KEY для столбца c_no, которое говорит о том, что значения в этом столбце должны быть уникальными, а неопределенные значения не допускаются. Такой столбец можно использовать для того, чтобы гарантированно отличить одну строку в таблице от других.

Полный список ограничений целостности для СУБД PostgreSQL можно найти по ссылке https://postgrespro.ru/docs/postgrespro/15/ddl-constraints.

Точный синтаксис команды CREATE TABLE (и любой другой SQLкоманды) можно посмотреть в документации к СУБД, а можно прямо в psql с помощью команды: `test=# \help CREATE TABLE`

Полный список команд покажет команда \help в консоли без параметров.

Наполнение таблиц

Добавим в созданную таблицу несколько строк с помощью оператора INSERT (postgrespro.ru/docs/postgresql/15/sql-insert). Обратите внимание на то, что мы опустили стандартное построчное приглашение с именем базы данных, в реальной консоли оно обязательно будет присутствовать):

```
test=> INSERT INTO courses(c_no, title, hours)
test-> VALUES ('CS301', 'Базы данных', 30),
test-> ('CS305', 'Сети ЭВМ', 60);
INSERT 0 2
```

Нам потребуется еще две таблицы: студенты и экзамены. Для каждого студента будем хранить его имя и год поступления; идентифицироваться он будет числовым номером студенческого билета.

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

Добавим в созданную таблицу информацию о трех студентах.

```
test=> INSERT INTO students(s_id, name, start_year)
test-> VALUES (1451, 'Анна', 2014),
test-> (1432, 'Виктор', 2014),
test-> (1556, 'Нина', 2015);
INSERT 0 3
```

Экзамен содержит оценку, полученную студентом по некоторой дисциплине. Таким образом, студенты и дисциплины связаны друг с другом отношением «многие ко многим» (часто обозначается как M:M): один студент может сдавать экзамены по многим дисциплинам, а экзамен по одной дисциплине могут сдавать много студентов.

Запись в таблице экзаменов идентифицируется совокупностью номера студенческого билета и номера курса. Такое ограничение целостности, относящее сразу к нескольким столбцам, определяется с помощью фразы

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

Кроме того, с помощью фразы REFERENCES мы определили 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, что все идентификаторы s_id, указанные в таблице экзаменов, соответствуют реальным студентам (то есть записям в таблице студентов), а номера c_no — реальным курсам. Таким образом, будет исключена возможность оценить несуществующего студента или поставить оценку 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)
test-> VALUES (1451, 'CS301', 5),
test-> (1556, 'CS301', 5),
test-> (1451, 'CS305', 5),
test-> (1432, 'CS305', 4);
INSERT 0 4
```

Выборка данных

Чтение данных из таблиц выполняется оператором SQL SELECT (postgrespro.ru/docs/postgresql/15/sql-select). Общий синтаксис оператора такой:

```
SELECT ([ALL | DISTINCT]< список полей>|*)
FROM <Список таблиц>
[WHERE <Предикат-условие выборки или соединения>]
[GROUP BY   <Список полей результатам]
[HAVING      <Предикат-условие для группы>]
[ORDER BY   <Список полей, по которым упорядочить вывод>]
```

Например, выведем только два столбца из таблицы courses:

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

Конструкция AS позволяет переименовать столбец, если это необходимо. 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 строки)
```

В результирующей выборке мы можем получить несколько одинаковых строк. Даже если все строки были различны в исходной таблице, дубликаты могут появиться, если выводятся не все столбцы:

```
test=> SELECT start_year FROM students;
 start_year
------------
       2014
       2014
       2015
(3 строки)
```

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

```
test=> SELECT DISTINCT start_year FROM students;
 start_year
------------
       2014
       2015
(2 строки)
```

Вообще после слова SELECT можно указывать любые выражения, например математические. А без фразы FROM результирующая таблица будет содержать одну строку. Например:

```
test=> SELECT 2+2 AS result;
 result
--------
      4
(1 строка)
```

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

```
test=> SELECT * FROM courses WHERE hours > 45;
 c_no  |  title   | hours
-------+----------+-------
 CS305 | Сети ЭВМ |    60
(1 строка)
```

Условие должно иметь логический тип. Например, оно может содержать отношения =, <> (или !=), >, >=, <, <=; 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 результирующую таблицу попадают только те строки, для которых условие

фильтрации истинно; если же значение ложно или не определено, строка отбрасывается.

При работе с неопределенными значениями следует учесть:

- 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 отношения IS NULL (IS NOT NULL) и IS DISTINCT FROM (IS NOT DISTINCT FROM), а также бывает удобно воспользоваться функцией coalesce.

Более подробно ознакомиться с функциями сравнения в PostgreSQL можно в документации: postgrespro.ru/doc/functions-comparison.

Соединения

Грамотно спроектированная реляционная база данных не содержит избыточных данных. 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 во фразе 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 строк)
```

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 во фразе WHERE условие соединения. Получим оценки по всем дисциплинам, сопоставляя курсы с теми экзаменами, которые проводились именно по данному курсу:

```
test=> SELECT courses.title, exams.s_id, exams.score
test-> FROM courses, exams
test-> 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 строки)
```

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

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

Этот пример показывает, что в результат не включаются строки исходной таблицы, для которых не нашлось пары в другой таблице: хотя условие наложено на дисциплины, но при этом исключаются и студенты, которые не сдавали экзамен по данной дисциплине. Чтобы в выборку попали все студенты, надо использовать внешнее соединение:

```
test=> SELECT students.name, exams.score
test-> FROM students
test-> LEFT JOIN exams
test-> ON students.s_id = exams.s_id
test-> AND exams.c_no = 'CS305';
  name  | score
--------+-------
 Анна   |     5
 Виктор |     4
 Нина   |
(3 строки)
```

В этом примере в результат добавляются 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 во фразе WHERE применяются к уже готовому результату соединений, поэтому, если вынести ограничение на дисциплины из условия соединения, Нина не попадет в выборку — ведь для нее exams.c_no не определен:

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

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.

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,
test-> (SELECT score
test(> FROM exams
test(> WHERE exams.s_id = students.s_id
test(> AND exams.c_no = 'CS305')
test-> FROM students;
  name  | score
--------+-------
 Анна   |     5
 Виктор |     4
 Нина   |
(3 строки)
```

Если скалярный подзапрос, использованный в списке выражений SELECT, не содержит ни одной строки, возвращается неопределенное значение (как в последней строке результата примера). 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 *
test-> FROM exams
test-> WHERE (SELECT start_year
test(> FROM students
test(> WHERE students.s_id = exams.s_id) > 2014;
 s_id | c_no  | score
------+-------+-------
 1556 | CS301 |     5
(1 строка)
```

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-нибудь оценки по указанному курсу:

```
test=> SELECT name, start_year
test-> FROM students
test-> WHERE s_id IN (SELECT s_id
test(> FROM exams
test(> WHERE c_no = 'CS305');
  name  | start_year
--------+------------
 Анна   |       2014
 Виктор |       2014
(2 строки)
```

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

```
test=> SELECT name, start_year
test-> FROM students
test-> WHERE s_id NOT IN
test-> (SELECT s_id FROM exams WHERE score = 5);
  name  | start_year
--------+------------
 Виктор |       2014
(1 строка)
```

Обратите внимание, что такой запрос вернет и всех студентов, не получивших вообще ни одной оценки.

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

```
test=> SELECT name, start_year
test-> FROM students
test-> WHERE NOT EXISTS (SELECT s_id
test(> FROM exams
test(> WHERE exams.s_id = students.s_id
test(> AND score = 5);
  name  | start_year
--------+------------
 Виктор |       2014
(1 строка)
```

Подробнее

Псевдонимы

В примерах выше мы уточняли имена столбцов названиями таблиц, чтобы избежать неоднозначности. Иногда этого недостаточно. Например, в запросе одна и та же таблица может участвовать два раза, или вместо таблицы в предложении FROM мы можем использовать безымянный подзапрос. В этих случаях после

смот

подзапроса можно указать произвольное имя, которое называется псевдонимом

Псевдонимы можно использовать и для обычных таблиц. Выведем имена студентов и их оценки по предмету «Базы данных»:

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

Здесь s — псевдоним таблицы, а ce — псевдоним подзапроса. Псевдонимы обычно выбирают так, чтобы они были короткими, но оставались понятными.

Тот же запрос можно записать и без подзапросов, например так:

```
test=> SELECT s.name, e.score
test-> FROM students s, courses c, exams e
test-> WHERE c.c_no = e.c_no
test-> AND c.title = 'Базы данных'
test-> AND s.s_id = e.s_id;
 name | score
------+-------
 Анна |     5
 Нина |     5
(2 строки)
```

Сортировка

Как уже говорилось, данные в таблицах не упорядочены, но часто бывает важно получить строки результата в строго определенном порядке. Для этого 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 *
test-> FROM exams
test-> 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 строки)
```

Здесь строки упорядочены сначала по возрастанию оценки, для совпадающих оценок — по возрастанию номера студенческого билета, а при совпадении первых двух ключей — 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-

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),
test-> avg(score)
test-> FROM exams;
 count | count |        avg
-------+-------+--------------------
     4 |     3 | 4.7500000000000000
(1 строка)
```

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

```
test=> SELECT c_no, count(*),
test-> count(DISTINCT s_id), avg(score)
test-> FROM exams
test-> GROUP BY c_no;
 c_no  | count | count |        avg
-------+-------+-------+--------------------
 CS301 |     2 |     2 | 5.0000000000000000
 CS305 |     2 |     2 | 4.5000000000000000
(2 строки)
```

Полный список агрегатных функций доступен в документации:

В запросах, использующих группировку, 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 можно также использовать столбцы таблицы-результата).

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

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

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

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
test-> SET hours = hours * 2
test-> 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
```

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

Транзакции

Давайте немного расширим нашу схему данных и распределим студентов по группам. При этом потребуем, чтобы у каждой группы в обязательном порядке был староста. Для этого создадим таблицу групп:

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

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

Теперь в таблице студентов нам необходим еще один столбец — номер группы, — о котором мы не подумали сразу. К счастью, в уже существующую таблицу можно добавить новый столбец:

```
test=> ALTER TABLE students
test-> 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
                    Таблица "public.students"
  Столбец   |   Тип   | Правило сортировки | Допустимость NULL | По умолчанию
------------+---------+--------------------+-------------------+--------------
 s_id       | integer |                    | not null          |
 name       | text    |                    |                   |
 start_year | integer |                    |                   |
 g_no       | text    |                    |                   |
Индексы:
    "students_pkey" PRIMARY KEY, btree (s_id)
Ограничения внешнего ключа:
    "students_g_no_fkey" FOREIGN KEY (g_no) REFERENCES groups(g_no)
Ссылки извне:
    TABLE "exams" CONSTRAINT "exams_s_id_fkey" FOREIGN KEY (s_id) REFERENCES students(s_id)
    TABLE "groups" CONSTRAINT "groups_monitor_fkey" FOREIGN KEY (monitor) REFERENCES students(s_id)
```

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

```
test=> \d
          Список отношений
 Схема  |   Имя    |   Тип   | Владелец
--------+----------+---------+-----------
 public | courses  | таблица | test_user
 public | exams    | таблица | test_user
 public | groups   | таблица | test_user
 public | students | таблица | test_user
(4 строки)
```

Создадим теперь группу «A-101» и поместим в нее всех студентов, а старостой сделаем Анну. Тут возникает затруднение. С одной стороны, мы не можем создать группу, не указав старосту. А с другой, как мы можем назначить Анну старостой, если она еще не входит в группу? Это привело бы к появлению в базе данных логически некорректных, несогласованных данных.

Мы столкнулись с тем, что две операции надо совершить одновременно, потому что ни одна из них не имеет смысла без другой. Такие операции, составляющие логически неделимую единицу работы, называются транзакцией. Начнем транзакцию:

```
test=> BEGIN;
BEGIN
test=*>
```

Затем добавим группу вместе со старостой. Поскольку мы не помним наизусть номер студенческого билета Анны, выполним запрос прямо в команде добавления строк:

```
test=> BEGIN;
BEGIN
test=*> INSERT INTO groups(g_no, monitor)
test-*> SELECT 'A-101', s_id
test-*> FROM students
test-*> WHERE name = 'Анна';
INSERT 0 1
test=*>
```

«Звездочка» в приглашении напоминает о незавершенной транзакции. Откройте теперь 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? Проверим в новом терминале:

```
postgres=# \c test
Вы подключены к базе данных "test" как пользователь "postgres".
```

Реализуем в нем запрос:

```
test=> SELECT * FROM groups;
 g_no | monitor
------+---------
(0 строк)
```

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
test=*>
```

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 строки)
```

А теперь завершим транзакцию, зафиксировав все сделанные изменения:

```
test=*> COMMIT;
COMMIT
test=>
```

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 строка)


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 строки)
```

СУБД дает несколько очень важных гарантий:

Во-первых, любая транзакция либо выполняется целиком (как в нашем примере), либо не выполняется совсем. Если бы в 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, когда фиксируются изменения транзакции, все ограничения целостности должны быть выполнены, иначе транзакция прерывается. В начале работы транзакции данные находятся в согласованном состоянии, и в конце своей работы транзакция оставляет их согласованными; это свойство так и называется — *согласованность*.

В-третьих, как мы убедились на примере, другие пользователи никогда не увидят несогласованные данные, которые транзакция еще не зафиксировала. 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 одновременном изменении одной и той же строки двумя разными процессами.

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 управления базами данных. Подробнее о транзакциях см. ialtransactions (и еще более подробно — postgrespro.ru/doc/mvcc).

Представления

С помощью команды CREATE VIEW можно создать представление запроса. Создаваемое представление лишено физической материализации, поэтому запрос будет выполняться при каждом обращении к представлению.

```
test=> CREATE VIEW young_stud AS
test->     SELECT *
test->     FROM students
test->     WHERE start_year = 2015;
CREATE VIEW
test=> select * from young_stud;
 s_id | name | start_year | g_no
------+------+------------+-------
 1556 | Нина |       2015 | A-101
(1 строка)
```

Детально с информацией о видах и способах организации представлений можно ознакомиться в документации: postgrespro.ru/docs/postgresql/15/sqlcreateview.