# Лабораторная работа № 1. Работа с СУБД PostgresSQL c помощью PSQL

**Цель работы:** Изучить возможности терминального клиента PSQL и способы применения основных команд при работе с СУБД.

**Краткие теоретические сведения**

Чтобы подключиться к серверу СУБД и выполнить какие-либо команды, требуется программа-клиент. Речь пойдет о терминальном клиенте psql, работа с которым происходит интерактивно в режиме командной строки.

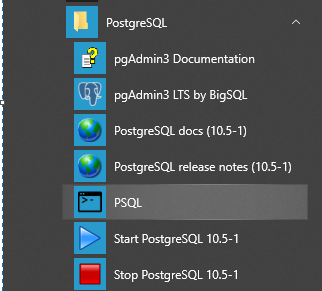
К сожалению, в наше время многие недолюбливают командную строку. Почему имеет смысл научиться с ней работать?

Во-первых, psql — стандартный клиент, входящий в любую сборку PostgreSQL, и поэтому он всегда под рукой. Безусловно, хорошо иметь настроенную под себя среду, но нет никакого резона оказаться беспомощным в незнакомом окружении.

Во-вторых, psql действительно удобен для повседневных задач по администрированию баз данных, для написания небольших запросов и автоматизации процессов, например, для периодической установки изменений программного кода на сервер СУБД. Он имеет собственные команды, позволяющие сориентироваться в объектах, хранящихся в базе данных, и удобно представить информацию из таблиц.

**Запуск PSQL**

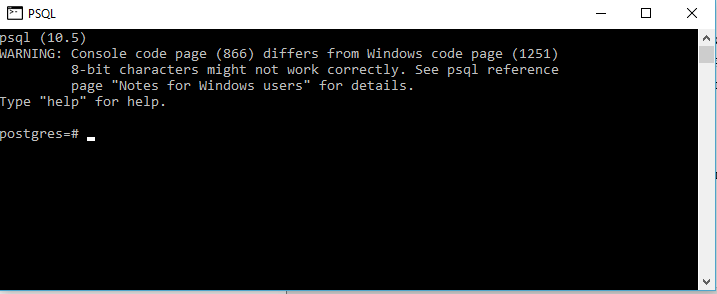
В Windows запустите программу «PSQL» из папки меню «Пуск», которую вы указали при установке (рис. 1.):



**Рис.1.** *Запуск PSQL*

В ответ на запрос введите пароль пользователя postgres, который вы указали при установке PostgreSQL.

В итоге вы увидите одинаковое приглашение postgres=#. «Postgres» здесь — имя базы данных, к которой вы сейчас подключены (рис.2). Один сервер PostgreSQL может одновременно обслуживать несколько баз данных, но одновременно вы работаете только с одной из них.



**Рис.2.** *Окно PSQL*

# Создание База данных

Давайте 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 (так что команду можно разбить на несколько строк).

Теперь переключимся на созданную базу:

**postgres=# \c test**

**You are now connected to database "test" as user "postgres". test=#**

Как вы 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.

Команд psql довольно много, и с некоторыми из них мы познакомимся чуть позже, а полный список с кратким описанием можно получить прямо сейчас: **test=# \?**

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

# Создание Таблиц

В реляционных СУБД данные представляются в виде таблиц. Заголовок таблицы определяет столбцы; собственно, данные располагаются в строках. Данные не упорядочены (в частности, нельзя полагаться на то, что строки хранятся в порядке их добавления в таблицу).

Для каждого столбца устанавливается тип данных; значения соответствующих полей строк должны удовлетворять этим типам. PostgreSQL располагает большим числом встроенных типов данных и возможностями для создания новых, но мы ограничимся несколькими из основных:

* **integer — целые числа;**
* **text — текстовые строки;**
* **boolean — логический тип, принимающий значения true (истина) или false (ложь).**

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.)

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 **PRIMARYKEY** 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 отличить одну строку в таблице от других.

Точный синтаксис команды CREATETABLE можно посмотреть в документации, либо попросить справку прямо в psql:

**test=# \help CREATE TABLE**

Такая справка есть по каждой команде SQL, а полный список команд легко получить с помощью \help без параметров.

# Наполнение таблиц

Добавим в созданную таблицу несколько строк:

**test=# INSERT INTO courses(c\_no, title, hours)**

**VALUES ('CS301', 'ППО1', 30),**

**('CS305', 'АСД', 60);**

INSERT 0 2

Если вам требуется массовая загрузка данных из внешнего источника, команда **INSERT** — не лучший выбор; посмотрите на специально предназначенную для этого команду COPY.

Для дальнейших примеров нам потребуется еще две таблицы: студенты и экзамены. Для каждого студента мы будем хранить его имя и год поступления; идентифицироваться он будет числовым 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, 'Айбек', 2017),**

**(1432, 'Мария', 2017),**

**(1556, 'Нина', 2018);**

INSERT 0 3

Экзамен содержит оценку, полученную студентом по некоторой дисциплине. 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 имени студента и номера курса. Такое ограничение целостности, относящее сразу к нескольким столбцам, определяется с помощью фразы **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 фразы REFERENCES мы определили два ограничения ссылочной целостности, называемые внешними ключами. Такие ключи 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)**

**VALUES (1451, 'CS301', 5),**

**(1556, 'CS301', 5),**

**(1451, 'CS305', 5),**

**(1432, 'CS305', 4);**

INSERT 0 4

# Задание

* Создайте базу данных согласно вашему проекту;
* Создайте таблицы БД;
* Заполните таблицы вашей БД.

# Лабораторная работа № 2. Выборка данных и формирование запросов в PostgresSQL

**Цель работы:** Изучение основных команд выборки данных и формирование запросов в клиенте PSQL.

# Выборка данных. Простые запросы

Чтение данных из таблиц выполняется оператором SELECT. Например, выведем два столбца из таблицы courses:

**test=# SELECT title AS course\_title, hours**

**FROM courses;**

course\_title | hours

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

ППО1 | 30

ОС | 60

(2 rows)

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 | ППО1 | 30

CS305 | ОС | 60

(2 rows)

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

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

2017

2017

2018

(3 rows)

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

**test=# SELECT DISTINCT start\_year FROM students;**

start\_year

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

2017

2018

(2 rows)

Подробнее смотрите в документации: [postgrespro.ru/ doc/sql-select.html#SQL-DISTINCT](https://postgrespro.ru/doc/sql-select.html#SQL-DISTINCT)

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

**test=# SELECT 2+2 AS result;**

result

--------

4

(1 row)

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

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

c\_no | title | hours

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

CS305 | ППО1 | 60

(1 row)

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.

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

# 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 | ППО1 | 30 | 1451 | CS301 | 5

CS305 | ОС | 60 | 1451 | CS301 | 5

CS301 | ППО1 | 30 | 1556 | CS301 | 5

CS305 | ОС | 60 | 1556 | CS301 | 5

CS301 | ППО1 | 30 | 1451 | CS305 | 5

CS305 | ОС | 60 | 1451 | CS305 | 5

CS301 | ППО1 | 30 | 1432 | CS305 | 4

CS305 | ОС | 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 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**

**FROM courses, exams**

**WHERE courses.c\_no = exams.c\_no;**

title | s\_id | score

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

ППО1 | 1451 | 5

ППО1| 1556 | 5

ОС | 1451 | 5

ОС | 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 «ОС»:

**test=# SELECT students.name, exams.score**

**FROM students**

**JOIN exams**

**ON students.s\_id = exams.s\_id AND exams.c\_no = 'CS305';**

name | score

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

Мария| 5

Айбек | 4

(2 rows)

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, независимо от того, сдавали они экзамен или нет, надо использовать операцию внешнего соединения:

**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

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

Мария| 5

Айбек | 4

Нина |

(3 rows)

В этом примере в результат добавляются строки из левой таблицы (поэтому операция называется **LEFTJOIN**), 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**

**FROM students**

**LEFT JOIN exams ON students.s\_id = exams.s\_id**

**WHERE exams.c\_no = 'CS305';**

name | score

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

Мария| 5

Айбек | 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.

# 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 одну строку и 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

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

Мария| 5

Айбек | 4

Нина |

(3 rows)

Если подзапрос, использованный в списке выражений **SELECT**, не содержит ни одной строки, возвращается неопределенное значение (как в последней строке результата примера).

Такие скалярные подзапросы можно использовать и в условиях фильтрации. Получим все экзамены, которые сдавали студенты, поступившие после 2017 года:

**test=# SELECT \***

**FROM exams**

**WHERE (SELECT start\_year**

**FROM students**

**WHERE students.s\_id = exams.s\_id) > 2017;**

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 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

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

Мария| 2017

Айбек | 2017

(2 rows)

Вариантом является отношение **NOTIN**, возвращающее противоположный результат. Например, список студентов, получивших только отличные оценки (то есть не получивших более низкие оценки):

**test=# SELECT name, start\_year FROM students**

**WHERE s\_id NOT IN (SELECT s\_id**

**FROM exams**

**WHERE score < 5);**

name | start\_year

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

Мария | 2017

Нина | 2018

(2 rows)

Другая возможность — 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**

**FROM students**

**WHERE NOT EXISTS (SELECT s\_id**

**FROM exams**

**WHERE exams.s\_id = students.s\_id AND score < 5);**

name | start\_year

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

Мария | 2017

Нина | 2018

(2 rows)

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 подзапроса можно указать произвольное имя, которое называется псевдонимом (**alias**). Псевдонимы можно использовать и для обычных таблиц.

Имена студентов и их оценки по предмету «Базы данных»:

**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

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

Мария | 5

Нина | 5

(2 rows)

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 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, но часто бывает важно получить строки результата в строго определенном порядке. Для этого используется предложение **ORDERBY** 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 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.

# 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:

**SELECT count(\*), count(DISTINCT s\_id), avg(score)**

**FROM exams;**

count | count | avg

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

4 | 3 | 4.7500000000000000

(1 row)

Аналогичную 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)

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:

**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

------

Мария

(1 row)

# 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 «ОС» в два раза:

**test=# UPDATE courses SET hours = hours \* 2**

**WHERE c\_no = 'CS301';**

UPDATE 1

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

**DELETE FROM exams WHERE score < 5;**

DELETE 1

# 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(**

**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 в таблице студентов нам необходимо еще одно поле — номер группы, — о которым мы не подумали при создании. 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 поля 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 «ПИ-17» и поместим в нее всех студентов, а старостой сделаем Марию.

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 тому, что в базе данных некоторое время (пусть и небольшое) находились бы логически некорректные, несогласованные данные.

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**

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

**test=# INSERT INTO groups(g\_no, monitor)**

**SELECT 'ПИ-17', s\_id**

**FROM students**

**WHERE name = 'Мария';**

INSERT 0 1

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 он 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 = 'ПИ-17';**

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 | Мария | 2017 |

1432 | Айбек | 2017 |

1556 | Нина | 2018 |

(3 rows)

А теперь завершим транзакцию, зафиксировав все 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

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

ПИ-17 |1451

(1 row)

**test=# SELECT \* FROM students;**

s\_id | name | start\_year | g\_no

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

1451 | Мария | 2017 | ПИ-17

1432 | Айбек | 2017 | ПИ-17

1556 | Нина | 2018 | ПИ-17

(3 rows)

СУБД гарантирует выполнение нескольких важных свойств.

Во-первых, 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. Таким образом, в начале 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.

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 управления базами данных.

Полезные команды psql

|  |  |
| --- | --- |
| **\?** | Справка по командам psql. |
| **\h** | Справка по SQL: список доступных команд или синтаксис конкретной команды. |
| **\x** | Переключает обычный табличный вывод (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. |
| **\d+ имя** | Еще более подробная информация по конкретному объекту. |
| **\timing on** | Показывать время выполнения операторов. |

И еще одна важная команда psql: для того, чтобы завершить сеанс работы, наберите **test=# \q**

# Задание

* В схеме с курсами и студентами предусмотрите возможность ведения занятий по курсу несколькими преподавателями. Экзамен может сдаваться не тому преподавателю, который вел занятия.
* Создайте схему базы, которую можно использовать для хранения исторических данных о погоде (температура, влажность, скорость ветра, время суток и т.п.).
* Создайте схему базы с информацией о наличии автомобилей в автомобильных салонах. Необходимо учитывать марку автомобиля, модель, год выпуска, адрес салона и т.п.