ЛЫСАНОВ Д.М, ИСАВНИН А.Г

СИСТЕМЫ УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ

Набережные Челны 2023

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования «Казанский (Приволжский) федеральный университет»

Набережночелнинский институт (филиал)

Кафедра Бизнес-информатики и математических методов в экономике

Системы управления базами данных.

Учебно-методическое пособие

Набережные Челны
2023 г.

УДК 510.872
ББК 22.171

Печатается по решению учебно-методической комиссии экономического отделения Набережночелнинского института (филиала) федерального государственного автономного образовательного учреждения высшего образования «Казанский (Приволжский) федеральный университет» от «05» сентября 2023 г. (протокол №5).

Рецензенты:
Доктор экономических наук, профессор А.Н. Макаров
Кандидат педагогических наук, доцент О.Ю.Герасимова

Лысанов Д.М, Исавнин А.Г Системы управления базами данных. Учебно-методическое пособие /Д.М.Лысанов, А.Г.Исавнин – Набережные Челны: Издательство Набережночелнинского института КФУ, 2023. – 85 с.

Учебно-методическое пособие предназначено для использования в учебном процессе студентами технических и экономических направлений, направлений ИКТ дневной, заочной и дистанционной форм обучения.

Содержание

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 разных языков программирования, а сейчас речь пойдет о терминальном клиенте psql, работа с которым происходит интерактивно в режиме командной строки.

К сожалению, многие недолюбливают командную строку. Почему же 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.

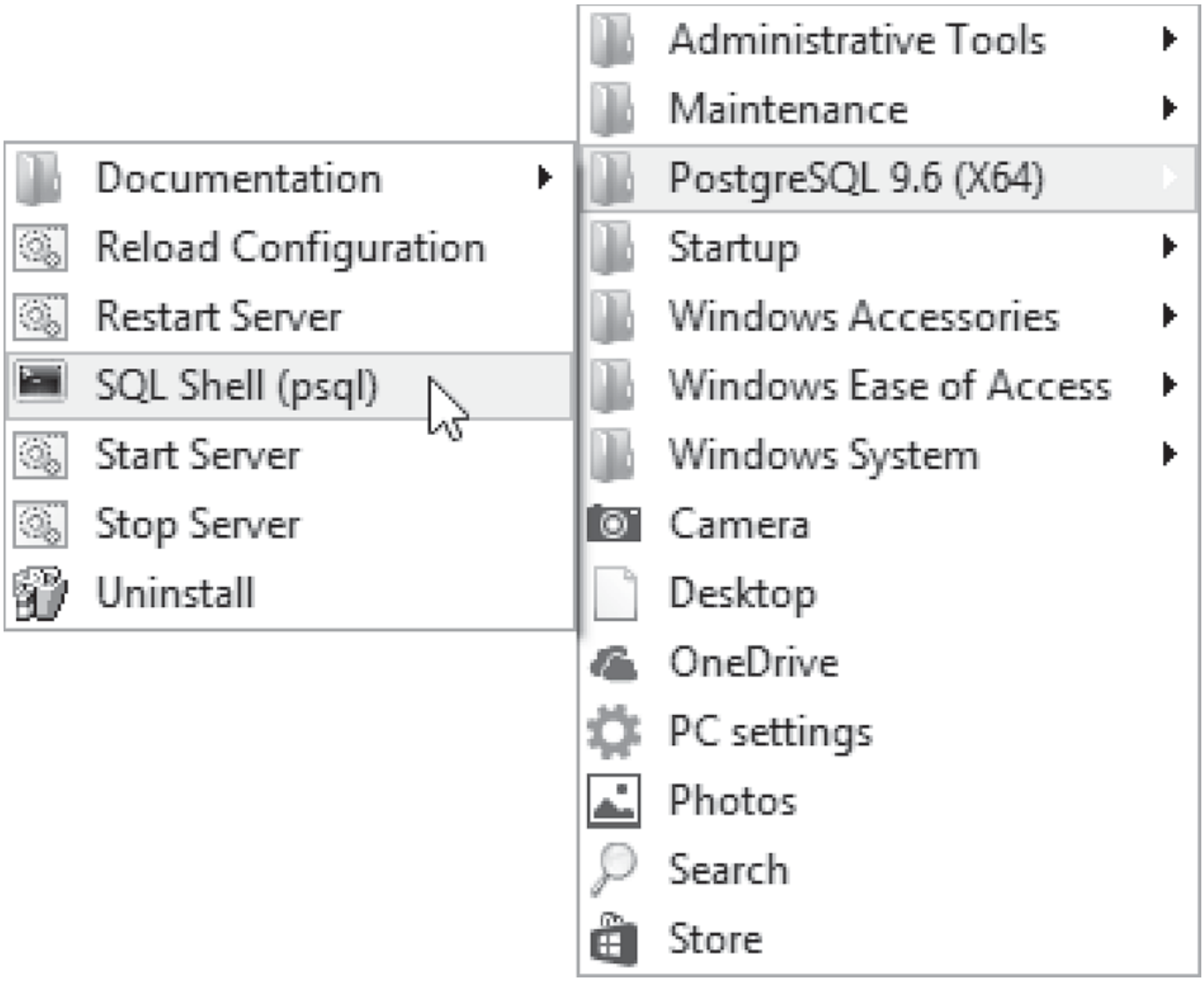
Во вторых, psql действительно удобен для повседневных задач по администрированию баз данных, для написания небольших запросов и автоматизации процессов, например, для периодической установки изменений программного кода на сервер СУБД. Он имеет собственные команды, позволяющие сориентироваться в объектах,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

Но если вы привыкли работать с графическими пользовательскими интерфейсами, попробуйте pgAdmin3 мы еще упомянем эту программу ниже или другие аналогичные продукты:
`wiki.postgresql.org/wiki/Community_Guide_to_PostgreSQL_GUI_Tools`

Чтобы запустить psql, в системе Linux выполните команду:

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

В Windows запустите программу «SQL Shell (psql)» из папки меню «Пуск», которую вы указали при установке:



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

Пользователи Windows могут столкнуться с неправильным отображением русских букв в терминале. Если вместо букв вы видите закорючки, сделайте следующее:

1. Закройте окно терминала и отредактируйте файл «`SQL Shell (psql)`» добавьте в него первой строкой команду:
   `chcp 1251`
2. Снова запустите psql и в свойствах окна измените шрифт с растрового на TrueType (обычно «Lucida Console»).

В итоге и в одной, и в другой операционной системе вы увидите одинаковое приглашение `postgres=#`. «Postgres» здесь имя базы данных, к которой вы сейчас подключены. Один сервер PostgreSQL может одновременно обслуживать несколько баз данных, но одновременно вы работаете только с одной из них.

Дальше мы будем приводить некоторые команды. Вводите только то, что выделено жирным шрифтом; приглашение и ответ системы на команду приведены исключительно для удобства.

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 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 (поэтому, если у вас открыт pgAdmin или другое графическое средство, пропускайте все, что начинается на косую черту, или попытайтесь найти аналог).

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`.

Таблицы

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 том порядке, в котором они добавлялись в таблицу).

Для каждого столбца устанавливается *тип данных,* а значения соответствующих полей строк должны удовлетворять этим типам. PostgreSQL располагает большим числом встроенных типов данных (полный список: `postgrespro.ru/doc/datatype.html`) и возможностями для создания новых, но мы ограничимся всего несколькими:

- `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 `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 отличить одну строку в таблице от других. Полный список ограничений целостности: `postgrespro.ru/doc/ddl constraints.html`.

Точный синтаксис команды `create table` 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', 'Базы данных', 30),
       ('CS305', 'Сети ЭВМ', 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`: `postgrespro.ru/doc/sql copy.html`.

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, 'Анна', 2014),
       (1432, 'Виктор', 2014),
       (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»: 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
```

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
--------------+-------
 Базы данных  |    30
 Сети ЭВМ     |    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 | Базы данных  |    30
 CS305 | Сети ЭВМ     |    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
------------
       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.html#SQL DISTINCT`

Вообще после слова `select` можно указывать не только столбцы, но и любые выражения. А если не указать фразу `from`, то 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 | Сети ЭВМ |    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`.

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

Соединения

Грамотно спроектированная 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 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
--------------+------+-------
 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 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
---------+-------
 Анна    |     5
 Виктор  |     4
 Нина    |
(3 rows)
```

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
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.html#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 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
---------+-------
 Анна    |     5
 Виктор  |     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 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 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)
```

Вариантом является отношение `not in`, 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
------+------------
 Анна |       2014
 Нина |       2015
(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);
```

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

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»:

```
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 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 `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 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_select.html#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)
```

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)
```

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

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
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.html#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.html

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.html

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,
  headman 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 у нас есть таблицы:

```
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 «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 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 неделимую единицу работы, называются *транзакцией*.

Начнем транзакцию:

```
test=# BEGIN;
BEGIN
```

Затем добавим группу вместе со старостой, использовав запрос в команде добавления строк:

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

Откройте теперь новое окно терминала и запустите еще один процесс psql: это будет сеанс, работающий параллельно с первым.

Увидит ли он наши изменения (чтобы не запутаться, команды второго сеанса мы будем показывать с отступом)?

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

Подробнее о транзакциях:
`postgrespro.ru/doc/tutorial transactions.html`
(и еще более подробно: `postgrespro.ru/doc/mvcc.html`).

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.

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: для того, чтобы завершить сеанс работы, наберите

```
test=# \q
```

Демонстрационная база данных

Описание

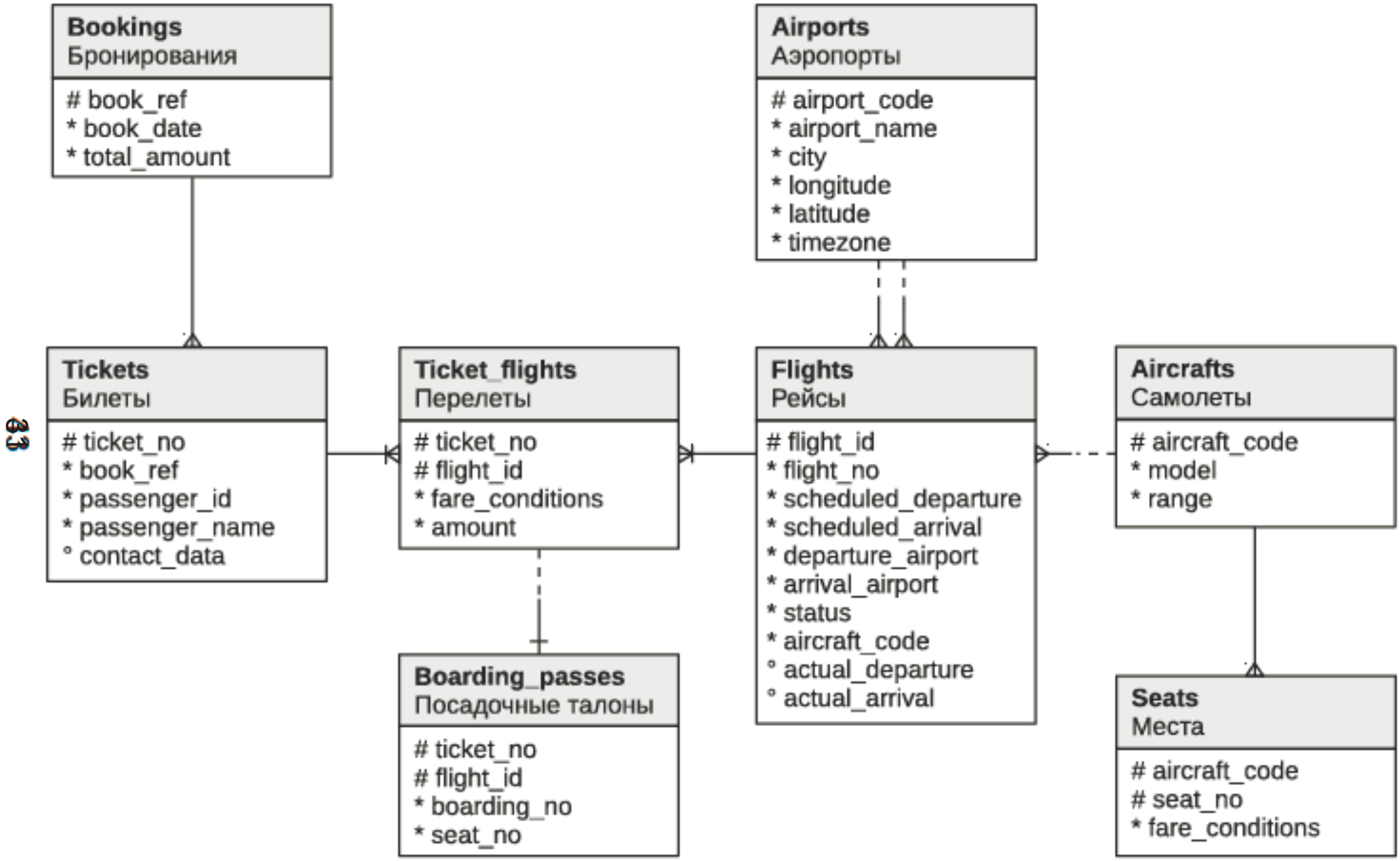
Общая информация

Чтобы двигаться дальше и учиться писать более сложные запросы, нам понадобится более серьезная база данных не три таблицы, а целых восемь, и наполнение ее данными. Схема такой базы данных изображена в виде диаграммы «сущность связи» на следующей странице.

В качестве предметной области мы выбрали авиаперевозки: будем считать, что речь идет о нашей (пока еще несуществующей) авиакомпании. Тем, кто хотя бы раз летал на самолетах, эта область должна быть понятна; в любом случае мы сейчас все объясним. Хочется отметить, что мы старались сделать схему данных как можно проще, не загромождая ее многочисленными деталями, но, в то же время, не слишком простой, чтобы на ней можно было учиться писать интересные и осмысленные запросы.

Итак, основной сущностью здесь является *бронирование (bookings)*.

В одно бронирование можно включить несколько пассажиров, каждому из которых выписывается отдельный *билет* (tickets). Как таковой пассажир не является отдельной сущностью: для простоты можно считать, что все пассажиры уникальны.


Bookings
Бронирования
book_ref
* book_date
* total_amount
Airports
Аэропорты
airport_code
* airport_name
* city
* longitude
* latitude
* timezone
Tickets
Билеты
ticket_no
* book_ref
* passenger_id
* passenger_name
° contact_data
Ticket_flights
Перелеты
ticket_no
flight_id
* fare_conditions
* amount
Flights
Рейсы
flight_id
* flight_no
* scheduled_departure
* scheduled_arrival
* departure_airport
* arrival_airport
* status
* aircraft_code
° actual_departure
° actual_arrival
Aircrafts
Самолеты
aircraft_code
* model
* range
Boarding_passes
Посадочные талоны
ticket_no
flight_id
* boarding_no
* seat_no
Seats
Места
aircraft_code
seat_no
* fare_conditions

Билет включает один или несколько *перелетов* (ticket_flights). Несколько перелетов могут включаться в билет в нескольких случаях:

1. Нет прямого рейса, соединяющего пункты отправления и назначения (полет с пересадками);
2. Взят билет «туда и обратно».

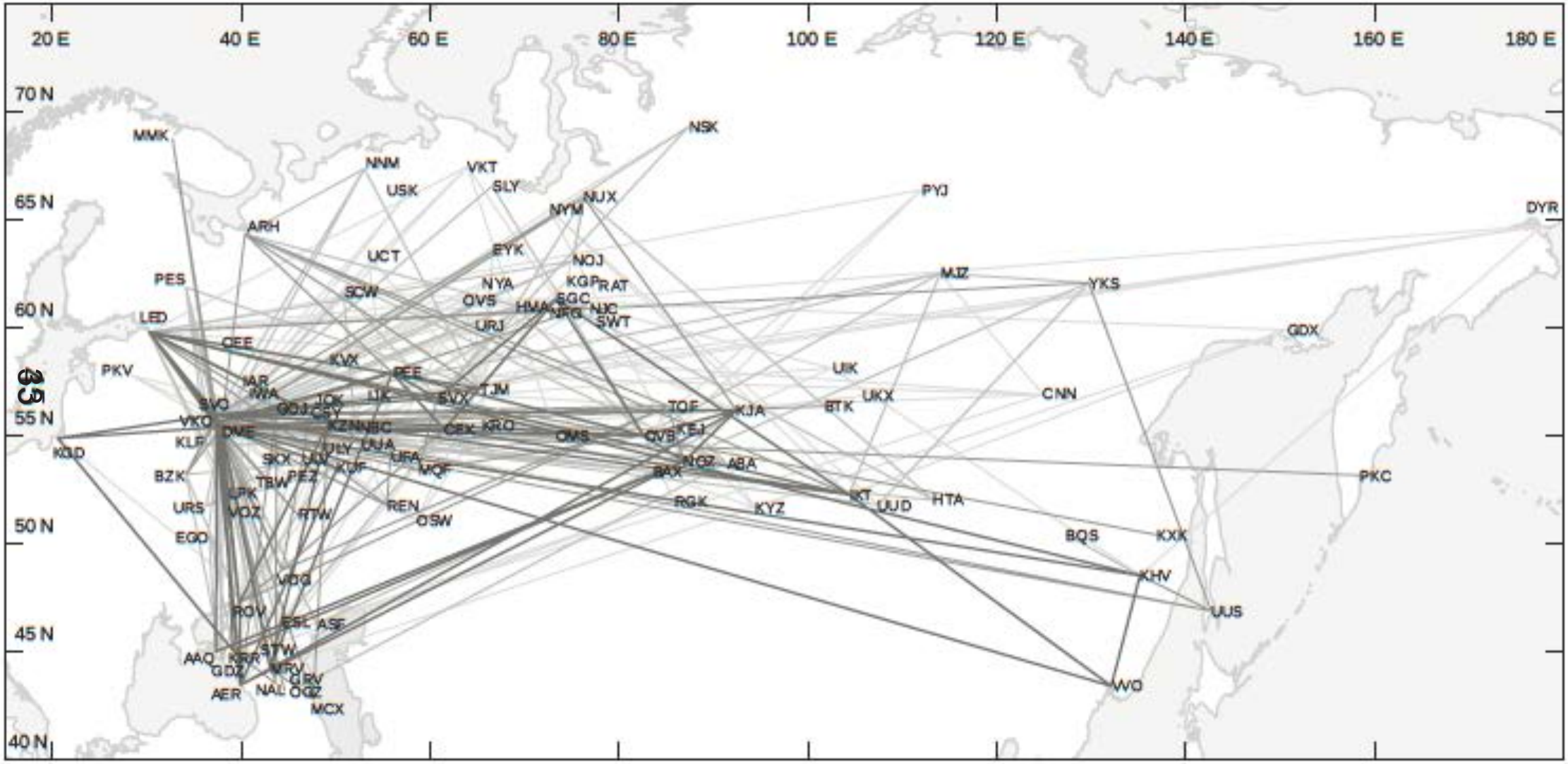
В схеме данных нет жесткого ограничения, но предполагается, что все билеты в одном бронировании имеют одинаковый набор перелетов.

Каждый *рейс* (flights) следует из одного *аэропорта* (airports) в другой. Рейсы с одним номером имеют одинаковые пункты вылета и назначения, но будут отличаться датой отправления

При регистрации на рейс пассажиру выдается *посадочный талон* (boarding_passes), в котором указано место в самолете. Пассажир может зарегистрироваться только на тот рейс, который есть у него в билете. Комбинация рейса и места в самолете должна быть уникальной, чтобы не допустить выдачу двух посадочных талонов на одно место.

Количество *мест* (seats) в самолете и их распределение по классам обслуживания зависит от модели *самолета* (aircrafts), выполняющего рейс. Предполагается, что каждая модель имеет только одну компоновку салона. Схема данных не контролирует, что места в посадочных талонах соответствуют имеющимся в самолете.

Далее мы подробно опишем каждую из таблиц, а также дополнительные представления и функции. Точное определение любой таблицы, включая типы данных и описание столбцов, вы всегда можете получить командой `\d+`.


20 E
40 E
60 E
80 E
100 E
120 E
140 E
160 E
180 E
70 N
65 N
60 N
55 N
50 N
45 N
40 N
MMK
NSK
NNM
VKT
USK
SLY
NUX
NYM
PYJ
DYR
ARH
UCT
EYK
NOJ
PES
NYA
KGP
RAT
MJZ
YKS
SCW
OVS
HMA
SGC
NJC
SWT
URJ
LED
GDX
CEE
KVX
PEE
PKV
UIK
IAR
IWA
TJM
SVX
UKX
CNN
SVO
TOF
KJA
BTK
VKO
DME
KZN
NBC
CEK
KRO
OMS
QVB
KEJ
KUD
KLF
ULY
ULA
SKX
UUA
UFA
NOZ
ABA
BZK
PEZ
KUF
MQF
BAX
PKC
TBW
LPK
URS
VOZ
RTW
REN
RGK
KYZ
UUD
HTA
OSW
BQS
KXK
EGO
KHV
VOG
ROV
ESL
ASF
UUS
STW
AAQ
KRR
GDZ
MRV
GRV
AER
NAL
OGZ
MCX

Бронирование

Намереваясь воспользоваться услугами нашей авиакомпании, пассажир заранее (book date, максимум за месяц до рейса) бронирует необходимые билеты. Бронирование идентифицируется своим номером (book ref, шестизначная комбинация букв и цифр).

Поле total amount хранит общую стоимость включенных в бронирование перелетов всех пассажиров.

Билет

Билет имеет уникальный номер (ticket no), состоящий из 13 цифр.

Билет содержит идентификатор пассажира (passenger id) номер документа, удостоверяющего личность, а также его фамилию и имя (passenger name) и контактную информацию (contact date).

Заметим, что ни идентификатор пассажира, ни имя не являются постоянными (можно поменять паспорт, можно сменить фамилию). Поэтому однозначно найти все билеты одного и того же пассажира невозможно. Для простоты можно считать, что все пассажиры уникальны.

Перелет

Перелет соединяет билет с рейсом и идентифицируется двумя их номерами.

Для каждого перелета указываются его стоимость (amount) и класс обслуживания (fare conditions).

Рейс

Естественный ключ таблицы рейсов состоит из двух полей номера рейса (flight no) и даты отправления

(scheduled departure). Чтобы сделать внешние ключи на эту таблицу компактнее, в качестве первичного используется суррогатный ключ (flight id).

Рейс всегда соединяет две точки аэропорты вылета (departure airport) и прибытия (arrival airport). Такое понятие, как «рейс с пересадками» отсутствует: если из одного аэропорта до другого нет прямого рейса, в билет просто включаются несколько необходимых рейсов.

У каждого рейса есть запланированные дата и время вылета (scheduled departure) и прибытия (scheduled arrival). Реальные время вылета (actual departure) и прибытия (actual arrival) могут отличаться: обычно не сильно, но иногда и на несколько часов, если рейс задержан.

Статус рейса (status) может принимать одно из следующих значений:

- **Scheduled**
  Рейс доступен для бронирования. Это происходит за месяц до плановой даты вылета; до этого запись о рейсе не существует в базе данных.
- **On Time**
  Рейс доступен для регистрации (за сутки до плановой даты вылета) и не задержан.
- **Delayed**
  Рейс доступен для регистрации (за сутки до плановой даты вылета), но задержан.
- **Departed**
  Самолет уже вылетел и находится в воздухе.
- **Arrived**
  Самолет прибыл в пункт назначения.
- **Cancelled**
  Рейс отменен.

Аэропорт

Аэропорт идентифицируется трехбуквенным кодом (airport code) и имеет название (airport name).

Название города (city) указывается как атрибут аэропорта; отдельной сущности для него не предусмотрено. Название можно использовать для того, чтобы определить аэропорты одного города. Также указывается широта (longitude), долгота (latitude) и часовой пояс (timezone).

Посадочный талон

При регистрации на рейс, которая возможна за сутки до плановой даты отправления, пассажиру выдается посадочный талон. Он идентифицируется так же, как и перелет номером билета и номером рейса.

Посадочным талонам присваиваются последовательные номера (boarding no) в порядке регистрации пассажиров на рейс (этот номер будет уникальным только в пределах данного рейса). В посадочном талоне указывается номер места (seat no).

Самолет

Каждая модель воздушного судна идентифицируется своим трехзначным кодом (aircraft code). Указывается также название модели (model) и максимальная дальность полета в километрах (range).

Место

Места определяют схему салона каждой модели. Каждое место определяется своим номером (seat no) и имеет закрепленный за ним класс обслуживания (fare conditions) Economy, Comfort или Business.

Представление для рейсов

Над таблицей flights создано представление flights_v, содержащее дополнительную информацию:

- расшифровку данных об аэропорте вылета (departure_airport, departure_airport_name, departure_city),
- расшифровку данных об аэропорте прибытия (arrival_airport, arrival_airport_name, arrival_city),
- местное время вылета (scheduled_departure_local, actual_departure_local),
- местное время прибытия (scheduled_arrival_local, actual_arrival_local),
- продолжительность полета (scheduled_duration, actual_duration).

Представление для маршрутов

Таблица рейсов содержит избыточность: из нее можно было бы выделить информацию о маршруте (номер рейса, аэропорты отправления и назначения), не зависящую от конкретных дат рейсов.

Именно такая информация и составляет материализованное представление routes.

Функция now

Демонстрационная база содержит временной «срез» данных так, как будто в некоторый момент была сделана резервная копия реальной системы. Например, если некоторый рейс имеет статус Departed, это означает, что в момент резервного копирования самолет вылетел и находился в воздухе.

Позиция «среза» сохранена в функции `bookings.now`. Ей можно пользоваться в запросах там, где в обычной жизни использовалась бы функция `now`.

Кроме того, значение этой функции определяет версию демонстрационной базы данных. Актуальная версия на момент подготовки этого выпуска книги от 13.10.2016.

Установка

Установка с сайта

База данных доступна в трех версиях, которые отличаются только объемом данных:

- `demo_small.zip` небольшая, данные по полетам за один месяц (21 МБ, размер БД 265 МБ),
- `demo_medium.zip` средняя, данные по полетам за три месяца (62 МБ, размер БД 666 МБ),
- `demo_big.zip` большая, данные по полетам за год (232 МБ, размер БД 2502 МБ).

Небольшая база годится для того, чтобы тренироваться писать запросы, и при этом не займет много места на диске. Если же вы хотите погрузиться в вопросы оптимизации, выберите большую базу, чтобы сразу понять, как ведут себя запросы на больших объемах данных.

Файлы содержат логическую резервную копию базы «demo», созданную утилитой pg dump. Имейте в виду, что если у вас уже существует база данных «demo», она будет удалена и создана заново при восстановлении из резервной копии. Владельцем базы «demo» станет тот пользователь СУБД, под которым выполнялось восстановление.

Чтобы установить демонстрационную базу данных в Linux, скачайте один из файлов, предварительно переключившись на пользователя postgres. Например, для базы небольшого размера:

```
$ sudo su - postgres
$ wget https://edu.postgrespro.ru/demo_small.zip
```

Затем выполните команду:

```
$ zcat demo_small.zip | psql
```

В системе Windows скачайте файл
`https://edu.postgrespro.ru/demo_small.zip`
любым веб браузером, дважды кликните на нем, чтобы открыть архив, и скопируйте файл `demo_small.sql` в каталог
`C:\Program Files\PostgreSQL\9.6`

Затем запустите psql (ярлык «SQL Shell (psql)») и выполните команду:

```
postgres# \i demo_small.sql
```

(Если файл не будет найден, проверьте свойство «Start in» ярлыка.)

Примеры запросов

Пара слов о схеме

Теперь, когда установка выполнена, запустите psql и подключитесь к демонстрационной базе:

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

Все интересующие нас объекты находятся в схеме bookings. Это означает, что имена таблиц, представлений и функций надо предварять указанием схемы, иначе они не будут найдены:

```
demo=# select * from aircrafts;
ERROR:  relation "aircrafts" does not exist
LINE 1: select * from aircrafts;
                      ^
```

```
demo=# select * from bookings.aircrafts;

 aircraft_code |        model         | range
---------------+----------------------+-------
 773           | Boeing 777 300       | 11100
 763           | Boeing 767 300       |  7900
 SU9           | Sukhoi SuperJet 100  |  3000
 320           | Airbus A320 200      |  5700
 321           | Airbus A321 200      |  5600
 319           | Airbus A319 100      |  6700
 733           | Boeing 737 300       |  4200
 CN1           | Cessna 208 Caravan   |  1200
 CR2           | Bombardier CRJ 200   |  2700
(9 rows)
```

Чтобы не приходилось каждый раз указывать имя схемы, установите параметр `search_path` следующим образом:

```
demo=# set search_path = bookings, public;
SET
```

После этого все таблицы и представления будут доступны по имени.

Однако для функции `bookings.now` все равно необходимо указывать схему, чтобы отличать ее от стандартной функции now:

```
demo=# select bookings.now();
          now
------------------------
 2016 10 13 17:00:00+03
(1 row)
```

Подробнее про управление схемами:
`postgrespro.ru/doc/ddl schemas.html`
и про установку конфигурационных параметров:
`postgrespro.ru/doc/config setting.html`

Простые запросы

Ниже мы покажем некоторые примеры задач на емонстрационной схеме. Большинство из них приведены вместе с решениями, остальные предлагается решить самостоятельно.

Задача. Кто летел позавчера рейсом Москва (SVO) Новосибирск (OVB) на месте 1A, и когда он забронировал свой билет?

```
SELECT t.passenger_name,
       b.book_date
FROM   bookings b
       JOIN tickets t
         ON t.book_ref = b.book_ref
       JOIN boarding_passes bp
         ON bp.ticket_no = t.ticket_no
       JOIN flights f
         ON f.flight_id = bp.flight_id
WHERE  f.departure_airport = 'SVO'
AND    f.arrival_airport = 'OVB'
AND    f.scheduled_departure::date =
       bookings.now()::date   INTERVAL '2 day'
AND    bp.seat_no = '1A';
```

«Позавчера» отсчитывается от `booking.now`, а не от текущей даты.

Задача. Сколько мест осталось незанятыми вчера на рейсе PG0404?

Решение. Задачу можно решить несколькими способами. Первый вариант использует конструкцию not exists, чтобы определить места, на которые нет посадочных талонов:

```
SELECT count(*)
FROM   flights f
       JOIN seats s
       ON s.aircraft_code = f.aircraft_code
WHERE  f.flight_no = 'PG0404'
AND    f.scheduled_departure::date =
       bookings.now()::date   INTERVAL '1 day'
AND    NOT EXISTS (
         SELECT NULL
         FROM   boarding_passes bp
         WHERE  bp.flight_id = f.flight_id
         AND    bp.seat_no = s.seat_no
       );
```

Второй вариант использует операцию вычитания множеств:

```
SELECT count(*)
FROM   (
  SELECT s.seat_no
  FROM   seats s
  WHERE  s.aircraft_code = (
       SELECT aircraft_code
       FROM   flights
       WHERE  flight_no = 'PG0404'
       AND    scheduled_departure::date =
              bookings.now()::date   INTERVAL '1 day'
  )
  EXCEPT
  SELECT bp.seat_no
  FROM   boarding_passes bp
  WHERE  bp.flight_id = (
       SELECT flight_id
       FROM   flights
       WHERE  flight_no = 'PG0404'
       AND    scheduled_departure::date =
              bookings.now()::date   INTERVAL '1 day'
  )
) t;
```

Какой вариант использовать, во многом зависит от личных предпочтений. Необходимо только учитывать, что выполняться такие запросы будут по разному, так что если важна производительность, то имеет смысл попробовать оба.

Задача. На каких маршрутах произошли самые длительные задержки рейсов? Выведите список из десяти «лидеров».

Решение. В запросе надо учитывать только рейсы, которые уже вылетели:

```
SELECT   f.flight_no,
         f.scheduled_departure,
         f.actual_departure,
         f.actual_departure – f.scheduled_departure
         AS delay
FROM     flights f
WHERE    f.actual_departure IS NOT NULL
ORDER BY f.actual_departure – f.scheduled_departure
         DESC
LIMIT    10;
```

Агрегатные функции

Задача. Какова минимальная и максимальная продолжительность полета для каждого из возможных рейсов из Москвы в Санкт Петербург, и сколько раз вылет рейса был задержан больше, чем на час?

Решение. Здесь удобно воспользоваться готовым представлением flights_v, чтобы не выписывать соединения таблиц. В запросе учитываем только уже выполненные рейсы.

```
SELECT   f.flight_no,
         scheduled_duration,
         min(actual_duration),
         max(actual_duration),
         sum(CASE
               WHEN f.actual_departure >
                    f.scheduled_departure +
                    INTERVAL '1 hour'
               THEN 1 ELSE 0
             END) delays
FROM     flights_v f
WHERE    f.departure_city = 'Москва'
AND      f.arrival_city = 'Санкт Петербург'
AND      f.status = 'Arrived'
GROUP BY f.flight_no,
         f.scheduled_duration;
```

Задача. Найдите самых дисциплинированных пассажиров, которые зарегистрировались на все рейсы первыми. Учтите только тех пассажиров, которые совершали минимум два рейса.

Решение. Используем тот факт, что номера посадочных талонов выдаются в порядке регистрации.

```
SELECT   t.passenger_name,
         t.ticket_no
FROM     tickets t
         JOIN boarding_passes bp
           ON bp.ticket_no = t.ticket_no
GROUP BY t.passenger_name,
         t.ticket_no
HAVING   max(bp.boarding_no) = 1
AND      count(*) > 1;
```

Задача. Сколько человек бывает включено в одно бронирование?

Решение. Сначала посчитаем количество человек в каждом бронировании, а затем количество бронирований для каждого количества человек.

```
SELECT   tt.cnt,
         count(*)
FROM     (
           SELECT   t.book_ref,
                    count(*) cnt
           FROM     tickets t
           GROUP BY t.book_ref
         ) tt
GROUP BY tt.cnt
ORDER BY tt.cnt;
```

Оконные функции

Задача. Для каждого билета выведите входящие в него перелеты вместе с запасом времени на пересадку на следующий рейс. Ограничьте выборку теми билетами, которые были забронированы неделю назад.

Решение. Чтобы не обращаться к одним и тем же данным два раза, удобно воспользоваться функционалом оконных функций.

```
SELECT tf.ticket_no,
       f.departure_airport,
       f.arrival_airport,
       f.scheduled_arrival,
       lead(f.scheduled_departure) OVER w
       AS next_departure,
       lead(f.scheduled_departure) OVER w
       f.scheduled_arrival AS gap
```

```
FROM   bookings b
       JOIN tickets t
         ON t.book_ref = b.book_ref
       JOIN ticket_flights tf
         ON tf.ticket_no = t.ticket_no
       JOIN flights f
         ON tf.flight_id = f.flight_id
WHERE  b.book_date =
       bookings.now()::date   INTERVAL '7 day'
WINDOW w AS (PARTITION BY tf.ticket_no
             ORDER BY f.scheduled_departure);
```

Глядя в результаты запроса, можно обратить внимание, что запас времени в некоторых случаях составляет несколько дней. Как правило, это билеты, взятые туда и обратно, то есть мы видим уже не время пересадки, а время нахождения в пункте назначения. Используя решение задачи в разделе «Массивы» ниже, можно учесть этот факт в запросе.

Задача. Какие сочетания имени и фамилии встречаются чаще всего и какую долю от числа всех пассажиров они составляют?

Решение.

```
SELECT   passenger_name,
         round( 100.0 * cnt / sum(cnt) over (), 2)
         AS percent
FROM     (
           SELECT   passenger_name,
                    count(*) cnt
           FROM     tickets
           GROUP BY passenger_name
         ) t
ORDER BY percent DESC;
```

Задача. Решите предыдущую задачу отдельно для имен и отдельно для фамилий.

Решение. Приведем вариант для имен.

```
WITH p AS (
  SELECT left(passenger_name,
              position(' ' in passenger_name))
         AS passenger_name
  FROM   tickets
)
SELECT   passenger_name,
         round( 100.0 * cnt / sum(cnt) over (), 2)
         AS percent
FROM     (
           SELECT   passenger_name,
                    count(*) cnt
           FROM     p
           GROUP BY passenger_name
         ) t
ORDER BY percent DESC;
```

Вывод: не стоит объединять в одном поле несколько значений, если вы собираетесь работать с ними по отдельности; по научному это называется «первой нормальной формой».

Массивы

Задача. В билете нет указания, в один ли он конец, или туда и обратно. Однако это можно вычислить, сравнив первый пункт отправления с последним пунктом назначения. Вывести для каждого билета аэропорты отправления и назначения без учета пересадок, и признак, взят ли билет туда и обратно.

Решение. Задачу можно решить, например, с помощью оконных функций. Но, пожалуй, проще всего свернуть список аэропортов на пути следования в массив с помощью агрегатной функции array_agg, и работать с ним. В качестве аэропорта назначения для билетов «туда обратно» мы выбираем средний элемент

массива, предполагая, что пути «туда» и «обратно» имеют одинаковое число пересадок.

```
WITH t AS (
  SELECT ticket_no,
         a,
         a[1]                   departure,
         a[cardinality(a)]      last_arrival,
         a[cardinality(a)/2+1] middle
  FROM   (
           SELECT t.ticket_no,
                  array_agg( f.departure_airport
                    ORDER BY f.scheduled_departure)
                  || (array_agg( f.arrival_airport
                    ORDER BY f.scheduled_departure
                    DESC))[1]
                  AS a
           FROM   tickets t
                  JOIN ticket_flights tf
                    ON (tf.ticket_no = t.ticket_no)
                  JOIN flights f
                    ON (f.flight_id = tf.flight_id)
           GROUP BY t.ticket_no
         ) t
)
SELECT t.ticket_no,
       t.a,
       t.departure,
       CASE
         WHEN t.departure = t.last_arrival
           THEN t.middle
         ELSE t.last_arrival
       END arrival,
       (t.departure = t.last_arrival) return_ticket
FROM   t;
```

В таком варианте таблица билетов просматривается только один раз. Массив аэропортов выводится исключительно для наглядности; на большом объеме данных имеет смысл убрать его из запроса.

Задача. Найти билеты, взятые туда и обратно, в которых путь «туда» не совпадает с путем «обратно».

Задача. Найдите такие пары аэропортов, рейсы между которыми в одну и в другую стороны отправляются по разным дням недели.

Решение. Часть задачи по построению массива дней недели уже фактически решена в материализованном представлении routes. Остается только найти пересечение:

```
SELECT r1.departure_airport,
       r1.arrival_airport,
       r1.days_of_week dow,
       r2.days_of_week dow_back
FROM   routes r1
       JOIN routes r2
       ON  r1.arrival_airport = r2.departure_airport
       AND r1.departure_airport = r2.arrival_airport
WHERE  NOT (r1.days_of_week && r2.days_of_week);
```

Рекурсивные запросы

Задача. Как с помощью минимального числа пересадок можно долететь из Усть Кута (UKX) в Нерюнгри (CNN), и какое время придется провести в воздухе?

Решение. Здесь фактически требуется найти кротчайший путь в графе. Это можно сделать с помощью приведенного ниже рекурсивного запроса.

Подробно про рекурсивные запросы вы можете посмотреть в документации:
`postgrespro.ru/doc/queries with.html`

```
WITH RECURSIVE p(
  last_arrival,
  destination,
  hops,
  flights,
  flight_time,
  found
) AS (
  SELECT a_from.airport_code,
         a_to.airport_code,
         array[a_from.airport_code],
         array[]::char(6)[],
         interval '0',
         a_from.airport_code = a_to.airport_code
  FROM   airports a_from,
         airports a_to
  WHERE  a_from.airport_code = 'UKX'
  AND    a_to.airport_code = 'CNN'
  UNION ALL
  SELECT r.arrival_airport,
         p.destination,
         (p.hops || r.arrival_airport)::char(3)[],
         (p.flights || r.flight_no)::char(6)[],
         p.flight_time + r.duration,
         bool_or(r.arrival_airport = p.destination)

         OVER ()
  FROM   p
         JOIN routes r
           ON r.departure_airport = p.last_arrival
  WHERE  NOT r.arrival_airport = ANY(p.hops)
  AND    NOT p.found
)
SELECT hops,
       flights,
       flight_time
FROM   p
WHERE  p.last_arrival = p.destination;
```

Зацикливание предотвращается проверкой по массиву пересадок hops.

Обратите внимание, что поиск происходит «в ширину», то есть первый же путь, который будет найден, будет кратчайшим по числу пересадок. Чтобы не перебирать остальные пути (которых может быть очень много), используется признак «маршрут найден» (found), который рассчитывается с помощью оконной функции bool_or.

Поучительно сравнить скорость выполнения этого запроса с более простым вариантом без флага.

Задача. Какое максимальное число пересадок потребуется, чтобы добраться из любого аэропорта в любой другой?

Решение. Запрос похож на предыдущий, но теперь мы начинаем с выборки «каждый аэропорт с каждым», и для каждой такой пары ищем кротчайший путь.

```
WITH RECURSIVE p(
  departure,
  last_arrival,
  destination,
  hops,
  found
) AS (
  SELECT a_from.airport_code,
         a_from.airport_code,
         a_to.airport_code,
         array[a_from.airport_code],
         a_from.airport_code = a_to.airport_code
  FROM   airports a_from,
         airports a_to
  UNION ALL
```

```
  SELECT p.departure,
         r.arrival_airport,
         p.destination,
         (p.hops || r.arrival_airport)::char(3)[],
         bool_or(r.arrival_airport = p.destination)
         OVER (PARTITION BY p.departure,
                            p.destination)
  FROM   p
         JOIN routes r
           ON r.departure_airport = p.last_arrival
  WHERE  NOT r.arrival_airport = ANY(p.hops)
  AND    NOT p.found
)
SELECT max(cardinality(hops) 1)
FROM   p
WHERE  p.last_arrival = p.destination;
```

Признак «маршрут найден» здесь необходимо рассчитывать отдельно для каждой пары аэропортов.

Задача. Найдите кротчайший путь из Усть Кута (UKX) в Нерюнгри (CNN) с точки зрения чистого времени перелетов (игнорируя время пересадок).

Подсказка: этот путь может оказаться не оптимальным по числу пересадок.

```
WITH RECURSIVE p(
  last_arrival,
  destination,
  hops,
  flights,
  flight_time,
  min_time
) AS (
```

```
  SELECT a_from.airport_code,
         a_to.airport_code,
         array[a_from.airport_code],
         array[]::char(6)[],
         interval '0',
         null::interval
  FROM   airports a_from,
         airports a_to
  WHERE  a_from.airport_code = 'UKX'
  AND    a_to.airport_code = 'CNN'
  UNION ALL
  SELECT r.arrival_airport,
         p.destination,
         (p.hops || r.arrival_airport)::char(3)[],
         (p.flights || r.flight_no)::char(6)[],
         p.flight_time + r.duration,
         least(
           p.min_time, min(p.flight_time+r.duration)
         FILTER (
           WHERE r.arrival_airport = p.destination)
         OVER ())
  FROM   p
         JOIN routes r
           ON r.departure_airport = p.last_arrival
  WHERE  NOT r.arrival_airport = ANY(p.hops)
  AND    p.flight_time + r.duration <
         coalesce(p.min_time, interval '1 year')
)
SELECT hops,
       flights,
       flight_time
FROM   (
         SELECT hops,
                flights,
                flight_time,
                min(min_time) OVER () min_time
         FROM   p
         WHERE  p.last_arrival = p.destination
       ) t
WHERE  flight_time = min_time;
```

Функции и расширения

Задача. Найти расстояние между Калининградом (KGD) и Петропавловском Камчатским (PKV).

Решение. У нас имеются координаты аэропортов. Чтобы рассчитать расстояние, можно воспользоваться расширением earthdistance (и перевести результат из милей в километры).

```
CREATE EXTENSION IF NOT EXISTS cube;
CREATE EXTENSION IF NOT EXISTS earthdistance;
SELECT round(
         (point(a_from.longitude,a_from.latitude) <@>
          point(a_to.longitude,a_to.latitude))
         * 1.609344
       )
FROM   airports a_from,
       airports a_to
WHERE  a_from.airport_code = 'KGD'
AND    a_to.airport_code = 'PKC';
```

Задача. Нарисуйте граф рейсов между аэропортами.

Дополнительные возможности

Полнотекстовый поиск

Несмотря на мощь языка запросов SQL, его возможностей не всегда достаточно для эффективной работы с данными. Особенно это стало заметно в последнее время, когда лавины данных, обычно плохо структурированных, заполнили хранилища информации. Изрядная доля Больших Данных приходится на тексты, плохо поддающиеся разбиению на поля баз данных. Поиск документов на естественных языках, обычно с сортировкой результатов по релевантности поисковому запросу, называют полнотекстовым поиском. В самом простом и типичном случае запросом считается набор слов, а соответствие определяется частотой слов в документе. Примерно таким поиском мы занимаемся, набирая фразу в поисковике Google или Яндекс.

Существует большое количество поисковиков, платных и бесплатных, которые позволяют индексировать всю вашу коллекцию документов и организовать вполне качественный поиск. В этих случаях индекс важнейший инструмент и ускоритель поиска не является частью базы данных. А это значит, что такие ценимые пользователями СУБД особенности, как синхронизация содержимого БД, транзакционность, доступ к метаданным и использование их для ограничения области поиска, организация безопасной политики доступа к документам и многое другое, оказываются недоступны.

Недостатки у все более популярных документо ориентированных СУБД обычно в той же области: у них есть развитые средства полнотекстового поиска, но безопасность и заботы о синхронизации для них не приоритетны. К тому же

обычно они (MongoDB, например) принадлежат классу NoSQL СУБД, а значит по определению лишены всей десятилетиями накопленной мощи SQL.

С другой стороны традиционные SQL СУБД имеют встроенные средства текстового поиска. Оператор `LIKE`, в былые времена активно использовавшийся для поиска в текстовых файлах, входит в стандартный синтаксис SQL. Но гибкость его явно недостаточна. В результате производителям СУБД приходилось добавлять собственные расширения к стандарту SQL. У PostgreSQL это операторы сравнения `ILIKE`, `~`, `~*`, но и они не решают всех проблем, так как не умеют учитывать грамматические вариации слов, не приспособлены для ранжирования и не слишком быстро работают.

Если говорить об инструментах собственно полнотекстового поиска, то важно понимать, что до их стандартизации пока далеко, в каждой реализации СУБД свой синтаксис и свои подходы. В этом контексте российский пользователь PostgreSQL получает немалые преимущества: расширения полнотекстового поиска для этой СУБД созданы российскими разработчиками, поэтому возможность прямого контакта со специалистами или даже посещение их лекций поможет углубиться в технологические детали, если в этом возникнет потребность. Здесь же мы ограничимся простыми примерами.

Для изучения возможностей полнотекстового поиска создадим еще одну таблицу в демонстрационной базе данных. Пусть это будут наброски конспекта лекций преподавателя курсов, разбитые на главы лекции:

```
test=# CREATE TABLE course_chapters(
  c_no text REFERENCES courses(c_no),
  ch_no text,
  ch_title text,
  txt text,
CONSTRAINT pkt_ch PRIMARY KEY(ch_no, c_no)
);
CREATE TABLE
```

Введем в таблицу тексты первых лекций по знакомым нам специальностям CS301 и CS305:

```
test=# INSERT INTO course_chapters(c_no, ch_no,
ch_title, txt) VALUES
('CS301', 'I', 'Базы данных',
'С этой главы начинается наше знакомство ' ||
'с увлекательным миром баз данных'),
('CS301', 'II', 'Первые шаги',
'Продолжаем знакомство с миром баз данных. ' ||
'Создадим нашу первую текстовую базу данных'),
('CS305', 'I', 'Локальные сети',
'С этих сетей начнется наше полное приключений ' ||
'путешествие в интригующий мир сетей');
INSERT 0 3
```

Проверим результат:

```
test=# SELECT ch_no AS no, ch_title, txt
FROM course_chapters;
 no |    ch_title    |              txt
----+----------------+-------------------------------
 I  | Базы данных    | С этой главы начинается наше
                       знакомство с увлекательным
                       миром баз данных
 II | Первые шаги    | Продолжаем знакомство с миром
                       баз данных. Создадим нашу
                       первую текстовую базу данных
 I  | Локальные сети | С этих сетей начнется наше
                       полное приключений путешествие
                       в интригующий мир сетей
(3 rows)
```

Найдем в таблице информацию по базам данных традиционными средствами SQL, то есть используя оператор LIKE:

```
test=# SELECT txt
FROM course_chapters
WHERE txt LIKE '%базы данных%';
```

Мы получим предсказуемый ответ: `0 строк`. Ведь `LIKE` не знает, что в родительном падеже следует искать «баз данных» или «базу данных» в творительном. Запрос

```
test=# SELECT txt
FROM course_chapters
WHERE txt LIKE '%базу данных%';
```

выдаст строку из главы II (но не I, где база в другом падеже):

```
                          txt
-------------------------------------------------------
 Продолжаем знакомство с миром баз данных. Создадим
 нашу первую текстовую базу данных
```

В PostgreSQL есть оператор `ILIKE`, который позволяет не заботиться о регистрах, а то бы пришлось еще думать и о прописных/строчных буквах. Конечно, в распоряжении знатока SQL есть и регулярные выражения (шаблоны поиска), составление которых занятие увлекательное, сродни искусству. Но когда не до искусства, хочется иметь инструмент, который думал бы за тебя.

Поэтому мы добавим к таблице глав еще один столбец со специальным типом данных `tsvector`:

```
test=# ALTER TABLE course_chapters
ADD txtvector tsvector;
test=# UPDATE course_chapters
SET txtvector=to_tsvector('russian',txt);
test=# SELECT txtvector FROM course_chapters;
                          txtvector
-------------------------------------------------------
 'баз':10 'глав':3 'дан':11 'знакомств':6 'мир':9
 'начина':4 'наш':5 'увлекательн':8
 'баз':5,11 'дан':6,12 'знакомств':2 'мир':4 'наш':8
 'перв':9 'продолжа':1 'создад':7 'текстов':10
 'интриг':10 'мир':11 'начнет':4 'наш':5 'полн':6
 'приключен':7 'путешеств':8 'сет':3,12 'эт':2
(3 rows)
```

Мы видим, что в строках а) слова сократились до их неизменяемых частей (лексем), б) появились цифры, означающие позицию вхождения слова в текст (видно, что некоторые слова вошли 2 раза), в) в строку не вошли предлоги (а также не вошли бы союзы и прочие не значимые для поиска единицы предложения так называемые стоп слова).

Можно начинать полнотекстовый поиск. Однако для более продвинутого поиска нам хотелось бы включить в поисковую область и названия глав. Причем, дабы подчеркнуть их важность, мы наделим их *весом* (важностью) при помощи функции `setweight`. Поправим таблицу:

```
test=# UPDATE course_chapters SET txtvector=
setweight(to_tsvector('russian',ch_title),'B')||' '||
setweight(to_tsvector('russian',txt),'D');
UPDATE 3
test=# SELECT txtvector FROM course_chapters;
                                                txtvector
----------------------------------------------------------------------------------------------------
 'баз':1B,12 'глав':5 'дан':2B,13 'знакомств':8
 'мир':11 'начина':6 'наш':7 'увлекательн':10
 'баз':7,13 'дан':8,14 'знакомств':4 'мир':6 'наш':10
 'перв':1B,11 'продолжа':3 'создад':9 'текстов':12
 'шаг':2B
 'интриг':12 'локальн':1B 'мир':13 'начнет':6 'наш':7
 'полн':8 'приключен':9 'путешеств':10 'сет':2B,5,14
 'эт':4
(3 rows)
```

У лексем появился относительный вес B и D (из четырех возможных A,B,C,D). Реальный вес мы будем задавать при составлении запросов. Это придаст им дополнительную гибкость.

Во всеоружии вернемся к поиску. Функции `to_tsvector` симметрична функция `to_tsquery`, приводящая символьное выражение к типу данных `tsquery`, который используют в запросах.

```
test=# SELECT ch_title
FROM course_chapters
WHERE txtvector @@
  to_tsquery('russian','базы & данные');

  ch_title
-------------
 Базы данных
 Первые шаги
(2 rows)
```

Можно убедиться, что 'база & данных' и другие грамматические вариации дадут тот же результат. Мы использовали оператор сравнения @@ (*две собаки*), выполняющий работу, аналогичную LIKE. К сожалению, синтаксис оператора не допускает выражение естественного языка с пробелами, такие как *база данных*, поэтому слова соединяются логическим оператором И.

Аргумент 'russian' указывает на конфигурацию, которую использует СУБД. Она определяет подключаемые словари и парсер программу разбора фразы. Словари можно менять или дополнять другими словарями. Словарь стеммер типа snowball (он оставляет в слове неизменяемую часть) «прошит» в СУБД и устанавливается всегда.

Введенные веса позволят вывести записи по результатам рейтинга:

```
test=# SELECT ch_title,
  ts_rank_cd('{0.1, 0.0, 1.0, 0.0}', txtvector, q)
FROM course_chapters,
  to_tsquery('russian','базы & данных') q
WHERE txtvector @@ q
ORDER BY ts_rank_cd DESC;

  ch_title   | ts_rank_cd
-------------+------------
 Базы данных |    1.11818
 Первые шаги |       0.22
(2 rows)
```

Массив {0.1, 0.0, 1.0, 0.0} задает веса. Это не обязательный аргумент функции ts_rank_cd, по умолчанию массив 0.1, 0.2, 0.4, 1.0 соответствует D, C, B, A. Вес слова повышает значимость найденной строки, помогает ранжировать результаты. В заключительном эксперименте модифицируем выдачу. Будем считать, что найденные слова мы хотим выделить жирным шрифтом в странице html. Функция ts_headline задает наборы символов, обрамляющих слово, а также минимальное и максимальное количество слов в строке:

```
test=# SELECT ts_headline(
  'russian',
  txt,
  to_tsquery('russian','мир'),
  'StartSel=<b>,StopSel=</b>,MaxWords=50,MinWords=5'
)
FROM course_chapters
WHERE to_tsvector('russian',txt) @@
  to_tsquery('russian','мир');
                    ts_headline
-----------------------------------------------------
 знакомство с увлекательным <b>миром</b> баз данных
 <b>миром</b> баз данных. Создадим нашу
 путешествие в интригующий <b>мир</b> сетей
(3 rows)
```

Для ускорения полнотекстового поиска используются специальные индексы GiST и GIN, отличные от обычных индексов в базах данных. Но они, как и многие другие полезные знания о полнотекстовом поиске, останутся вне рамок этого краткого руководства пользователя.

Более подробно о полнотекстовом поиске можно узнать в главе 12 документации PostgreSQL 9.6 www.postgrespro.ru/docs/textsearch.html

Работа с JSON и JSONB

Реляционные базы данных, использующие SQL, создавались с большим запасом прочности: первой заботой их потребителей была целостность и безопасность данных, а объемы информации были несравнимы с современными. Когда появилось новое поколение СУБД NoSQL, сообщество призадумалось: куда более простая структура данных (вначале это были прежде всего огромные таблицы с всего двумя колонками: ключ значение) позволяла ускорить поиск на порядки. Они могли обрабатывать небывалые объемы информации и легко масштабировались, вовсю используя параллельные вычисления. В NoSQL базах не было необходимости хранить информацию по строкам, а хранение по столбцам для многих задач позволяло еще больше ускорить и распараллелить вычисления.

Когда прошел первый шок, стало понятно, что для большинства реальных задач простенькой структурой не обойтись. Стали появляться сложные ключи, потом группы ключей. Реляционные СУБД не желали отставать от жизни и начали добавлять возможности, типичные для NoSQL. Поскольку в реляционных СУБД изменение схемы данных связано с большими вычислительными издержками, оказался как никогда кстати новый тип данных JSON. Изначально он предназначался для JS программистов, в том числе для AJAX приложений, отсюда JS в названии. Он как бы брал сложность добавляемых данных на себя, позволяя создавать сложнные линейные и иерархические структуры объекты, добавление которых не требовало пересчета всей базы. Тем, кто делал приложения, уже не было необходимости модифицировать схему базы данных. Синтаксис JSON похож на XML своим строгим соблюдением иерархии данных. JSON достаточно гибок для того, чтобы работать с разнородной, иногда непредсказуемой структурой данных.

Допустим, в нашей демо базе студентов появилась возможность ввести личные данные: запустили анкету, расспросили преподавателей. В анкете не обязательно заполнять все пункты, а некоторые из них включают графу «другое» и «добавьте о себе данные по вашему усмотрению». Если бы мы добавили в базу новые данные в привычной манере, то в многочисленных появившихся столбцах или дополнительных таблицах было бы большое количество пустых полей. Но еще хуже то, что в будущем могут появиться новые столбцы, а тогда придется существенно переделывать всю базу. Но мы решим эту проблему, используя JSON и появившийся позже JSONB, в котором данные хранятся в экономичном бинарном виде, и который, в отличие от JSON, приспособлен к созданию индексов, ускоряющих поиск иногда на порядки.

Создадим таблицу с объектами JSON:

```
test=# CREATE TABLE student_details(
  de_id int,
  s_id int REFERENCES students(s_id),
  details json,
  CONSTRAINT pk_d PRIMARY KEY(s_id, de_id)
);
```

```
test=# INSERT INTO student_details(de_id, s_id,
details) VALUES
(1, 1451,
'{
  "достоинства": "отсутствуют",
  "недостатки": "неумеренное употребление мороженого"
}'
),
(2, 1432,
'{
  "хобби":
    {
      "гитарист":
        {
          "группа": "Постгрессоры",
          "гитары":["страт","телек"]
        }
    }
}'
),
(3, 1556,
'{
  "хобби": "косплей",
  "достоинства":
    {
      "мать-героиня":
        {
          "Вася": "м",
          "Семен": "м",
          "Люся": "ж",
          "Макар": "м",
          "Саша":"сведения отсутствуют"
        }
    }
}'
),
(4, 1451,
'{
  "статус": "отчислена"
}'
);
```

Проверим, все ли данные на месте. Для удобства соединим таблицы student details и students при помощи конструкции where, ведь в новой таблице отсутствуют имена студентов:

```
test=# SELECT s.name, sd.details
FROM student_details sd, students s
WHERE s.s_id=sd.s_id;
```

```
  name   |                    details
---------+-----------------------------------------------
 Анна    | {                                             +
         |   "достоинства": "отсутствуют",               +
         |   "недостатки": "неумеренное употребление
                                              мороженого"+
         | }
 Виктор  | {                                             +
         |   "хобби":                                    +
         |     {                                         +
         |       "гитарист":                             +
         |         {                                     +
         |           "группа": "Постгрессоры",           +
         |           "гитары":["страт","телек"]          +
         |         }                                     +
         |     }                                         +
         | }
 Нина    | {                                             +
         |   "хобби": "косплей",                         +
         |   "достоинства":                              +
         |     {                                         +
         |       "мать героиня":                         +
         |         {                                     +
         |           "Вася": "м",                        +
         |           "Семен": "м",                       +
         |           "Люся": "ж",                        +
         |           "Макар": "м",                       +
         |           "Саша":"сведения отсутствуют"       +
         |         }                                     +
         |     }                                         +
         | }
 Анна    | {                                             +
         |   "статус": "отчислена"                       +
         | }
(4 rows)
```

Допустим, нас интересуют записи, где есть информация о достоинствах студентов. Мы можем обратиться к содержанию ключа «достоинство», используя специальный оператор >>:

```
test=# SELECT s.name, sd.details
FROM student_details sd, students s
WHERE s.s_id=sd.s_id
AND sd.details->>'достоинства' IS NOT NULL;
  name  |                    details
--------+-----------------------------------------------
 Анна   | {                                             +
        |   "достоинства": "отсутствуют",               +
        |   "недостатки": "неумеренное употребление
                                              мороженого"+
        | }
 Нина   | {                                             +
        |   "хобби": "косплей",                         +
        |   "достоинства":                              +
        |     {                                         +
        |       "мать героиня":                         +
        |         {                                     +
        |           "Вася": "м",                        +
        |           "Семен": "м",                       +
        |           "Люся": "ж",                        +
        |           "Макар": "м",                       +
        |           "Саша":"сведения отсутствуют"       +
        |         }                                     +
        |     }                                         +
        | }
(2 rows)
```

Мы убедились, что 2 записи имеют отношение к достоинствам Анны и Нины, однако такой ответ нас вряд ли удовлетворит, так как на самом деле достоинства Анны «отсутствуют». Скорректируем запрос:

```
test=# SELECT s.name, sd.details
FROM student_details sd, students s
WHERE s.s_id=sd.s_id
AND sd.details->>'достоинства' IS NOT NULL
AND sd.details->>'достоинства'!='отсутствуют';
```

Убедитесь, что этот запрос оставит в списке только Нину, обладающую реальными, а не отсутствующими достоинствами.

Но такой способ срабатывает не всегда. Попробуем найти, на каких гитарах играет музыкант Витя:

```
test=# SELECT sd.de_id, s.name, sd.details
FROM student_details sd, students s
WHERE s.s_id=sd.s_id
AND sd.details->>'гитары' IS NOT NULL;
```

Запрос ничего не выдаст. Дело в том, что соответствующая пара ключ значение находится внутри иерархии JSON, вложена в пары более высокого уровня:

```
 Виктор | {                                                  +
        |   "хобби":                                         +
        |     {                                              +
        |       "гитарист":                                  +
        |         {                                          +
        |           "группа": "Постгрессоры",                +
        |           "гитары":["страт","телек"]               +
        |         }                                          +
        |     }                                              +
        | }
```

Чтобы добраться до гитар, воспользуемся оператором «#>» и пустимся по иерархии с «хобби»:

```
test=# SELECT sd.de_id, s.name,
  sd.details#>'{хобби,гитарист,гитары}'
FROM student_details sd, students s
WHERE s.s_id=sd.s_id
AND sd.details#>'{хобби,гитарист,гитары}'IS NOT NULL;
```

и убедимся, что Виктор фанат фирмы Fender:

```
 de_id |  name   |     ?column?
-------+---------+------------------
     2 | Виктор | ["страт","телек"]
```

У типа данных JSON есть младший брат: JSONB. Буква «B» подразумевает бинарный (а не текстовый) способ хранения данных. Такие данные можно плотно упаковать и поиск по ним быстрее. Последнее время JSONB используется намного чаще, чем JSON.

```
test=# ALTER TABLE student_details
ADD details_b jsonb;
test=# UPDATE student_details
SET details_b=to_jsonb(details);
test=# SELECT de_id, details_b
FROM student_details;
```

```
 de_id |                    details_b
-------+-----------------------------------------------
     1 | {"недостатки": "неумеренное употребление
       мороженого", "достоинства": "отсутствуют"}
     2 | {"хобби": {"гитарист": {"гитары": ["страт",
       "телек"], "группа": "Постгрессоры"}}}
     3 | {"хобби": "косплей", "достоинства": {"мать
       героиня": {"Вася": "м", "Люся": "ж", "Саша":
       "сведения отсутствуют", "Макар": "м",
       "Семен": "м"}}}
     4 | {"статус": "отчислена"}
(4 rows)
```

Можно заметить, что, кроме иной формы записи, изменился порядок значений в парах: Саша, сведения о которой, как мы помним, отсутствует, заняла теперь место в списке перед Макаром. Это не недостаток JSONB относительно JSON, а особенность хранения информации.

Для работы с JSONB набор операторов больше. Один из полезнейших операторов `@>` оператор вхождения в объект. Он напоминает `#>` для JSON.

Найдем запись, где упоминается дочь матери героини Люся:

```
test=# SELECT s.name, jsonb_pretty(sd.details_b)
FROM student_details sd, students s
WHERE s.s_id=sd.s_id
AND sd.details_b @>
    '{"достоинства":{"мать-героиня":{}}}';
```

```
 name |                    jsonb_pretty
------+-------------------------------------------------
 Нина | {                                               +
      |     "хобби": "косплей",                         +
      |     "достоинства": {                            +
      |         "мать-героиня": {                       +
      |             "Вася": "м",                        +
      |             "Люся": "ж",                        +
      |             "Саша": "сведения отсутствуют",     +
      |             "Макар": "м",                       +
      |             "Семен": "м"                        +
      |         }                                       +
      |     }                                           +
      | }
(1 row)
```

Мы использовали функцию `jsonb_pretty()`, которая форматирует вывод типа JSONB.

Или можно воспользоваться функцией `jsonb_each()`, разворачивающей пары ключ значение:

```
test=# SELECT s.name, jsonb_each(sd.details_b)
FROM student_details sd, students s
WHERE s.s_id=sd.s_id
AND sd.details_b @>
    '{"достоинства":{"мать-героиня":{}}}';
 name |                   jsonb_each
------+-------------------------------------------------
 Нина | (хобби,"""косплей""")
 Нина | (достоинства,"{""мать-героиня"": {""Вася"":
      ""м"", ""Люся"": ""ж"", ""Саша"": ""сведения
      отсутствуют"", ""Макар"": ""м"", ""Семен"":
      ""м""}}")
(2 rows)
```

Между прочим, вместо имени ребенка Нины в запросе было оставлено пустое место «{}». Такой синтаксис добавляет гибкости процессу разработки реальных приложений.

Но главное, пожалуй, возможность в JSONB создавать индексы, поддерживающие операторы @>, обратный ему <@ и многие другие. Среди поддерживающих JSONB есть полезнейший

для поиска обратный индекс GIN. Для JSON индексы не поддерживаются, поэтому для приложений с серьезной нагрузкой как правило лучше выбирать JSONB, а не JSON.

Подробнее о типах JSON и JSONB и функциях для работы с ними можно узнать на странице документации PostgreSQL 9.6

`www.postgrespro.ru/docs/datatype json`
и
`www.postgrespro.ru/docs/functions json`

PostgreSQL для приложения

Отдельный пользователь

В предыдущей главе мы подключались к серверу баз данных под пользователем postgres, единственным существующим сразу после установки СУБД. Но postgres обладает правами суперпользователя, поэтому приложению не следует использовать его для подключения к базе данных. Лучше создать нового пользователя и сделать его владельцем отдельной базы данных тогда его права будут ограничены этой базой.

```
postgres=# CREATE USER app password 'p@ssw0rd';
CREATE ROLE
postgres=# CREATE DATABASE appdb owner app;
CREATE DATABASE
```

Подробнее про пользователей и привилегии:
postgrespro.ru/doc/user manag.html
и postgrespro.ru/doc/ddl priv.html.

Чтобы подключиться к новой базе данных и работать с ней от имени созданного пользователя, выполните команду:

```
postgres=# \c appdb app localhost 5432
Password for user app: ***
You are now connected to database "appdb" as user
"app" on host "127.0.0.1" at port "5432".
appdb=>
```

В команде указываются последовательно имя базы данных (appdb), имя пользователя (app), узел (localhost

или 127.0.0.1) и номер порта (5432). Обратите внимание, что подсказка изменилась: вместо «решетки» (#) теперь отображается символ «больше» (>) решетка указывает на суперпользователя по аналогии с пользователем root в юниксе.

Со своей базой данных пользователь app может работать без ограничений. Например, можно создать в ней таблицу:

```
appdb=> CREATE TABLE greeting(s text);
CREATE TABLE
appdb=> INSERT INTO greeting VALUES ('Привет, мир!');
INSERT 0 1
```

Удаленное подключение

В нашем примере клиент и СУБД находятся на одном и том же компьютере. Разумеется, вы можете установить PostgreSQL на выделенный сервер, а подключаться к нему с другой машины (например, с сервера приложений). В этом случае вместо localhost надо указать адрес вашего сервера СУБД. Но этого недостаточно: по умолчанию из соображений безопасности PostgreSQL допускает только локальные подключения.

Чтобы подключиться к базе данных снаружи, необходимо отредактировать два файла.

Во первых, файл основных настроек postgresql.conf (расположение конфигурационных файлов мы обсуждали при установке). Найдите строку, определяющую, какие сетевые интерфейсы слушает PostgreSQL:

```
#listen_addresses = 'localhost'
```

и замените ее на:

```
listen_addresses = '*'
```

Во вторых, файл настроек аутентификации `pg_hba.conf`. Допишите в конец файла следующую строку:

```
host appdb app all md5
```

Это разрешит доступ к базе данных `appdb` пользователю `app` с любого адреса при указании верного пароля.

Подробнее о настройках аутентификации: `postgrespro.ru/doc/client authentication.html`

Проверка связи

Для того, чтобы обратиться к PostgreSQL из программы на каком либо языке программирования, необходимо использовать подходящую библиотеку и установить драйвер СУБД.

Ниже приведены простые примеры для нескольких популярных языков, которые помогут быстро проверить соединение с базой данных. Приведенные программы намеренно содержат только минимально необходимый код для запроса к базе данных; в частности, не предусмотрена никакая обработка ошибок. Не стоит рассматривать эти фрагменты как пример для подражания.

Если вы используете ОС Windows, не забудьте в окне Command Prompt сменить шрифт с растрового на True Type (например, на «Lucida Console») и выполнить команды:

```
C:\> chcp 1251
Active code page: 1251
C:\> set PGCLIENTENCODING=WIN1251
```

PHP

В языке PHP работа с PostgreSQL организована с помощью специального расширения. В Linux кроме самого PHP нам потребуется пакет с этим расширением:

```
$ sudo apt-get install php5-cli
$ sudo apt-get install php5-pgsql
```

PHP для Windows можно установить с сайта `windows.php.net/download`. Расширение для PostgreSQL уже входит в комплект, но в файле php.ini необходимо найти и раскомментировать строку (убрать точку с запятой):

```
;extension=php_pgsql.dll
```

Пример программы (`test.php`):

```
<?php
  $conn = pg_connect('host=localhost port=5432 ' .
                     'dbname=appdb user=app ' .
                     'password=p@ssw0rd') or die;
  $query = pg_query('select * from greeting') or die;
  while ($row = pg_fetch_array($query)) {
    echo $row[0].PHP_EOL;
  }
  pg_free_result($query);
  pg_close($conn);
?>
```

Выполняем:

```
$ php test.php
Привет, мир!
```

Расширение для PostgreSQL описано в документации: `php.net/manual/ru/book.pgsql.php`

Perl

В языке Perl работа с базами данных
организована с помощью интерфейса DBI.
Сам Perl предустановлен в Debian и Ubuntu, так что
дополнительно нужен только драйвер:

```
$ sudo apt-get install libdbd-pg-perl
```

Существует несколько сборок Perl для Windows, которые
перечислены на сайте `www.perl.org/get.html`.
ActiveState Perl и Strawberry Perl уже включают необходимый
для PostgreSQL драйвер.

Пример программы (`test.pl`):

```
use DBI;
my $conn = DBI >connect(
  'dbi:Pg:dbname=appdb;host=localhost;port=5432',
  'app','p@ssw0rd') or die;
my $query = $conn >prepare('select * from greeting');
$query >execute() or die;
while (my @row = $query >fetchrow_array()) {
  print @row[0]."\n";
}
$query >finish();
$conn >disconnect();
```

Выполняем:

```
$ perl test.pl
Привет, мир!
```

Интерфейс описан в документации:
`metacpan.org/pod/DBD::Pg`

Python

В языке Python для работы с PostgreSQL обычно используется библиотека psycopg (название обычно произносится как «сайко пи джи»). В Debian и Ubuntu язык Python версии 2 предустановлен, так что нужен только драйвер:

```
$ sudo apt-get install python-psycopg2
```

(Если вы используете Python 3, установите пакет python3 psycopg2.)

Python для Windows можно скачать с сайта `www.python.org`.

Библиотека psycopg доступна на сайте проекта `initd.org/psycopg` (выберите версию, соответствующую установленной версии Python), там же находится необходимая документация.

Пример программы (`test.py`):

```
import psycopg2
conn = psycopg2.connect(
  host='localhost', port='5432', database='appdb',
  user='app', password='p@ssw0rd')
cur = conn.cursor()
cur.execute('select * from greeting')
rows = cur.fetchall()
for row in rows:
  print row[0]
conn.close()
```

Выполняем:

```
$ python test.py
Привет, мир!
```

Java

В языке Java работа с базой данных организована через интерфейс JDBC. Предполагая, что установлен JDK 1.7:

```
$ sudo apt-get install openjdk-7-jdk
```

нам дополнительно потребуется пакет с драйвером JDBC:

```
$ sudo apt-get install libpostgresql-jdbc-java
```

JDK для Windows можно скачать с сайта `www.oracle.com/technetwork/java/javase/downloads`.

Драйвер JDBC доступен на сайте `jdbc.postgresql.org` (выберите версию, соответствующую установленной версии JDK), там же находится вся необходимая документация.

Пример программы (`Test.java`):

```
import java.sql.*;
public class Test {
  public static void main(String[] args)
  throws SQLException {
    Connection conn = DriverManager.getConnection(
      "jdbc:postgresql://localhost:5432/appdb",
      "app", "p@ssw0rd");
    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery(
      "select * from greeting");
    while (rs.next()) {
      System.out.println(rs.getString(1));
    }
    rs.close();
    st.close();
    conn.close();
  }
}
```

Компилируем и выполняем, указывая путь к классу драйверу JDBC (в Windows пути разделяются не двоеточием, а точкой с запятой):

```
$ javac Test.java
$ java -cp .:/usr/share/java/postgresql-jdbc4.jar \
Test
Привет, мир!
```

pgAdmin3

pgAdmin графическое средство для администрирования PostgreSQL, ставшее стандартом де факто. Программа упрощает основные задачи администрирования, отображает объекты баз данных, позволяет выполнять запросы SQL.

Установка

Чтобы установить pgAdmin на Windows, воспользуйтесь нашей сборкой, доступной по адресу `www.postgrespro.ru/windows`. Установщик задает несколько вопросов; можно оставить значения, предлагаемые по умолчанию.

В системах Debian или Ubuntu просто установите пакет pgadmin3:

```
$ sudo apt-get install pgadmin3
```

или возьмите самую свежую версию с сайта `www.pgadmin.org`.

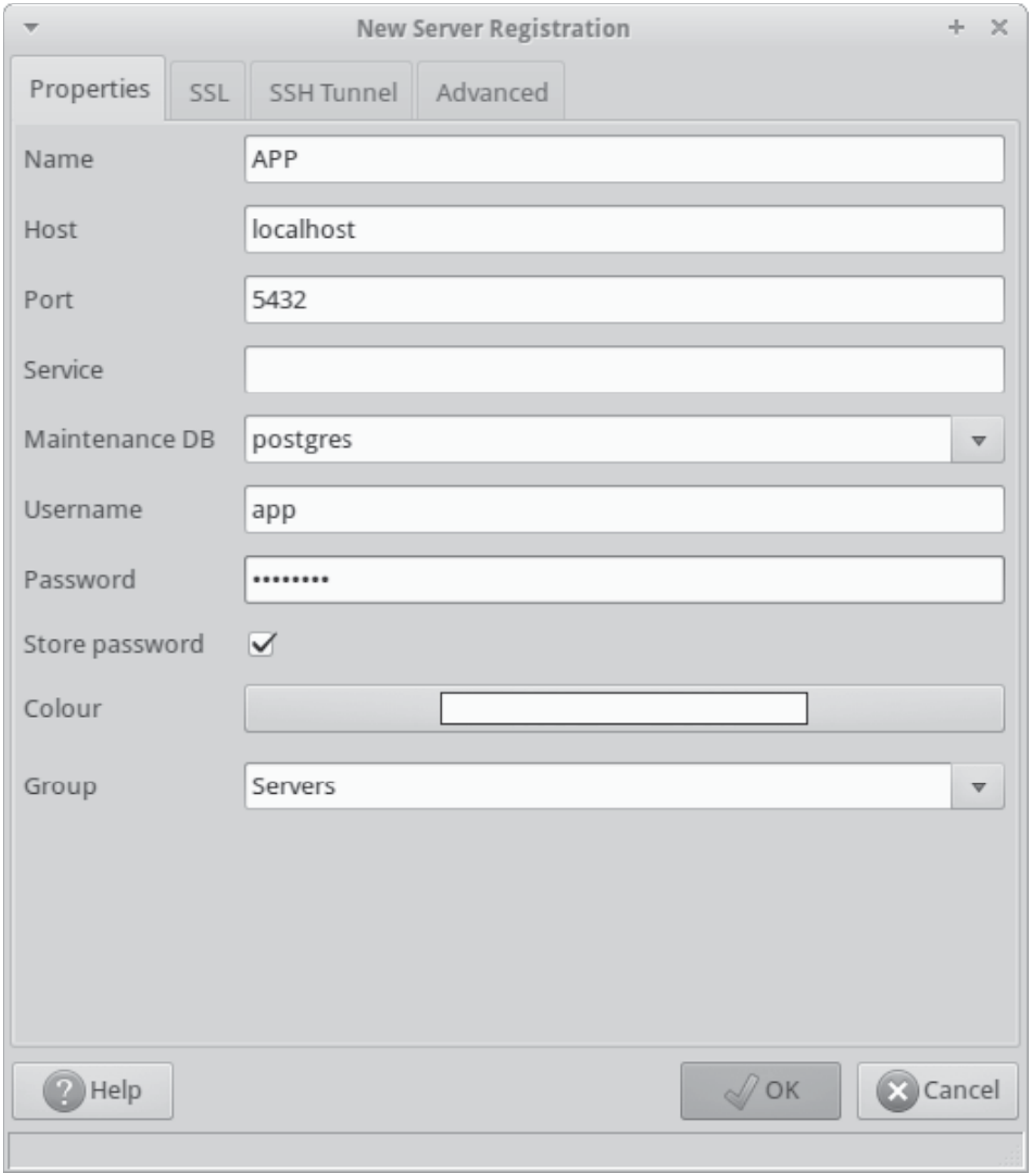
Возможности

Язык интерфейса

Вы можете в любой момент поменять язык интерфейса программы, выбрав в меню **File Options... (Файл Параметры...)** и изменив настройку **Miscellaneous User Interface (Разное Интерфейс)**. После этого требуется перезапуск программы.

Подключение к серверу

В первую очередь настроим подключение к серверу. Выберите в меню **File** **Add Server... (Файл** **Добавить сервер...)** и в появившемся окне введите произвольное имя для соединения **(Name)**, имя узла **(Host)**, порт **(Port)**, имя пользователя **(Username)** и пароль **(Password)**. Если не хотите вводить пароль каждый раз вручную, отметьте флажок **Store password (Сохранять пароль)**.

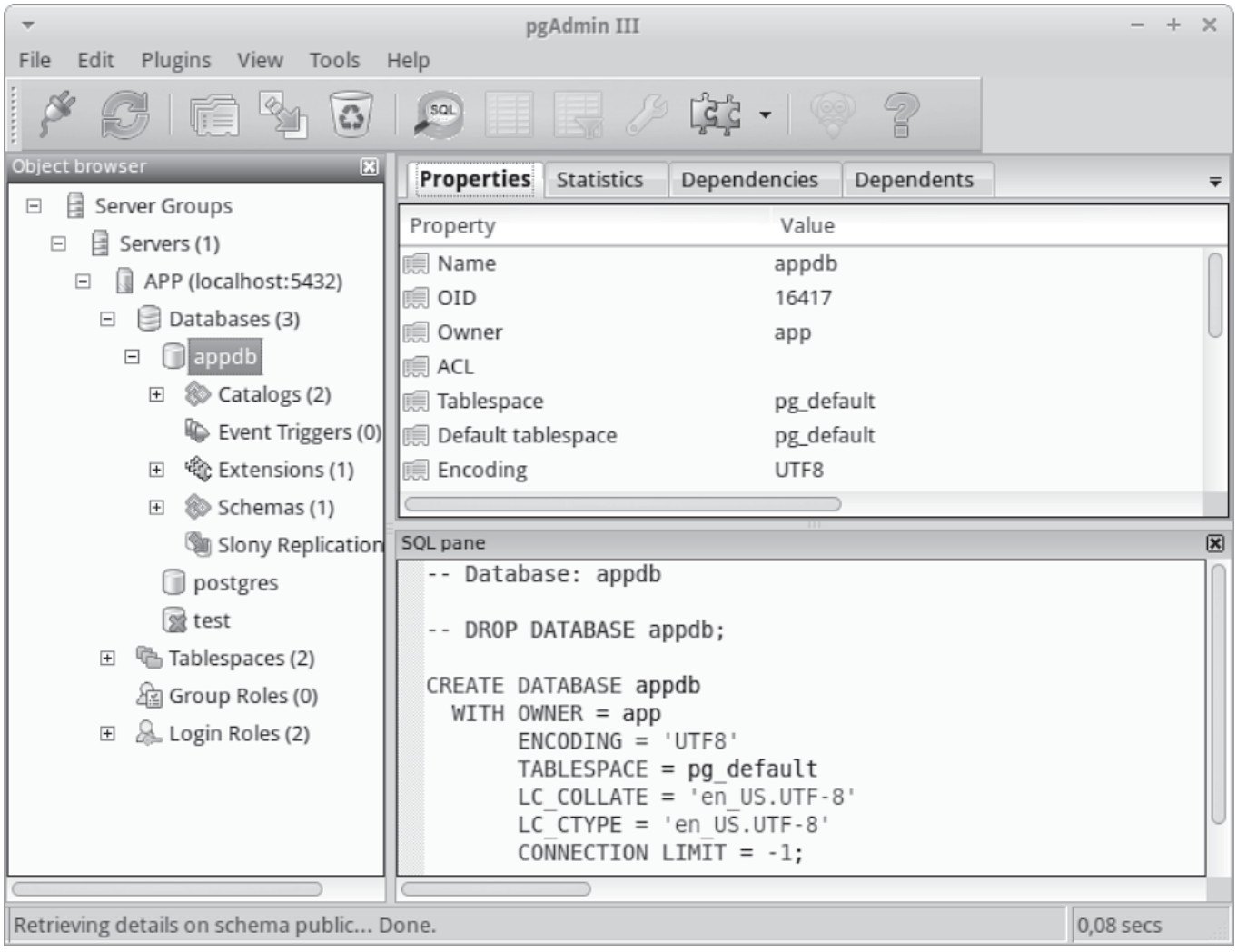


При нажатии на кнопку **OK** программа проверит доступность сервера с указанными параметрами и запомнит новое подключение.

Может оказаться, что ваша версия pgAdmin не поддерживает версию установленной базы данных. В таком случае при создании подключения будет выведено предупреждение, которые вы в большинстве случаев можете проигнорировать.

Навигатор

В левой части основного окна находится навигатор объектов. Разворачивая пункты списка, вы можете спуститься до сервера, который мы назвали APP. Ниже будут перечислены созданные в нем базы данных: `appdb` мы создали для проверки подключения к PostgreSQL из разных языков программирования, `test` когда знакомились с SQL, а база `postgres` создается автоматически при установке СУБД.



Развернув пункт **Schemas (Схемы)** для базы данных `appdb`, можно обнаружить и созданную нами ранее таблицу `greetings`, посмотреть ее столбцы, ограничения целостности, индексы, триггеры и т. п.

Не всегда бывает удобно разворачивать многочисленные пункты в навигаторе, чтобы добраться до интересующего нас объекта. Если известно имя или хотя бы часть имени объекта, его можно найти, выбрав в меню **Edit Search objects... (Правка Поиск объектов...)**; при этом в навигаторе должна быть выделена нужная база данных.

В появившемся окне введите, например, «greet%», при необходимости ограничьте поиск конкретным типом (таблицы, колонки и т. п.) и нажмите кнопку **Find (Найти)**. Если теперь выбрать в списке результатов нужный объект, навигатор автоматически переключится на него.

В правой части окна выводится справочная информация о выбранном объекте:

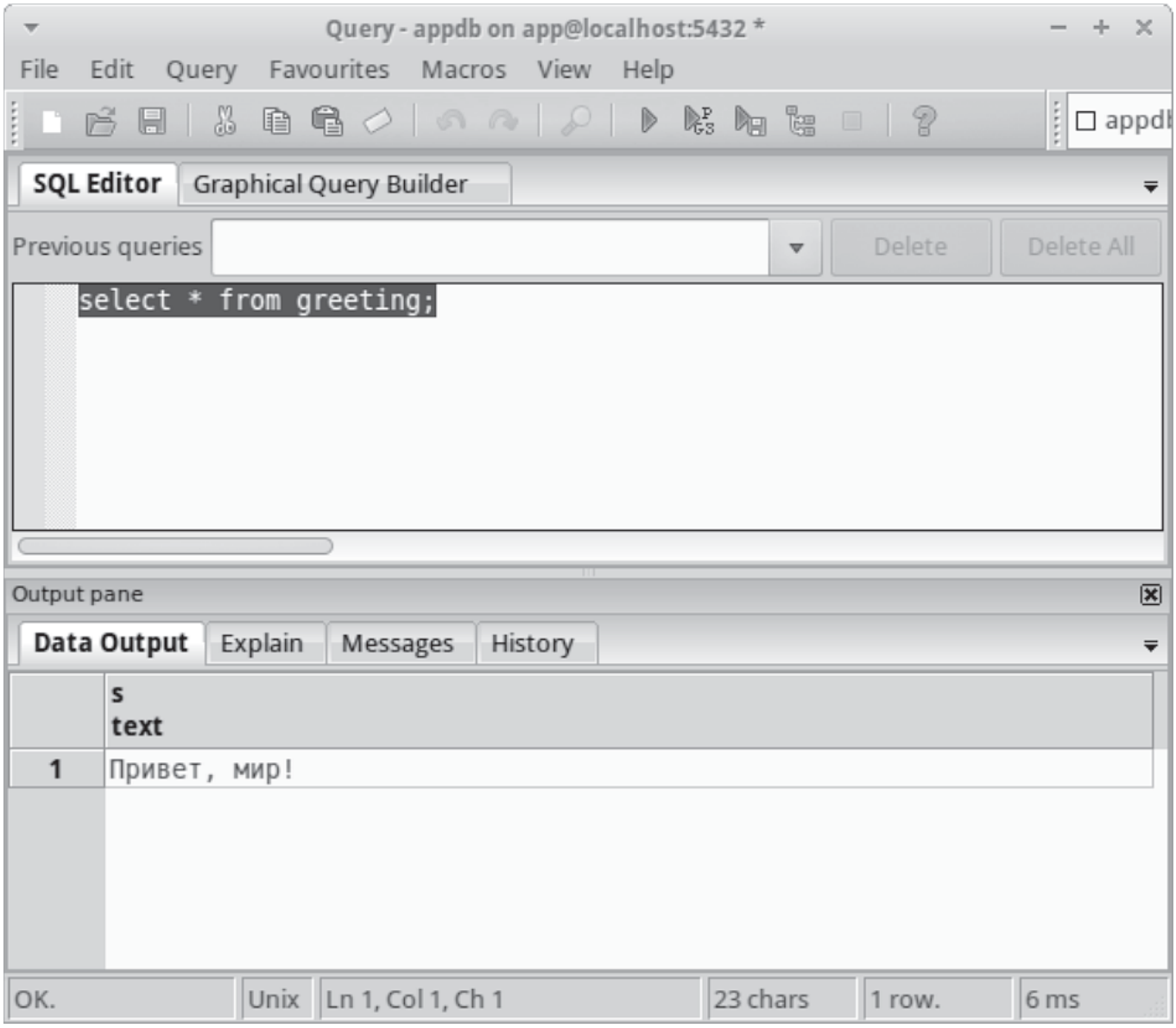
- Свойства зависят от типа объекта (например, для столбца будет показан тип его данных);
- Статистика эта информация используется системой для построения планов выполнения запросов и может рассматриваться администратором СУБД для анализа ситуации;
- Зависимости между объектами.

Также внизу правой части отображается команда SQL, с помощью которой можно создать данный объект.

Выполнение запросов

Чтобы выполнить запрос, откройте окно SQL, выбрав в меню **Tools Query tool (Инструменты Инструмент запросов)**.

Введите запрос в верхней части окна на вкладке **SQL Editor (Редактор SQL)** и нажмите F5. Внизу окна на вкладке **Data Output (Вывод данных)** появится результат.



Вы можете вводить следующий запрос на новой строке, не стирая предыдущий; просто выделите нужный фрагмент кода перед тем, как нажимать F5. Таким образом история ваших действий всегда будет у вас перед глазами обычно это удобнее, чем искать нужный запрос в списке **Previous queries (Предыдущие запросы).**

Другое

С другими возможностями pgAdmin вы можете познакомиться на сайте продукта `www.pgadmin.org` или в справочной системе самой программы.

Список использованных источников

1. Гарсиа-Молина, Г. Системы баз данных. Полный курс: пер. с англ. / Гектор Гарсиа-Молина, Джеффри Д. Ульман, Дженнифер Уидом. - М.: Вильямс, 2003. - 1088 с.: ил.

2. Гроff, Дж. Р. SQL. Полное руководство: пер. с англ. / Джеймс Р. Гроff, Пол Н. Вайнберг, Эндрю Дж. Оппель. - 3-е изд. - М.: Вильямс, 2015. - 960 с.: ил.

3. Дейт, К. Дж. Введение в системы баз данных: пер. с англ. / Крис Дж. Дейт. - 8-е изд. - М.: Вильямс, 2005. - 1328 с.: ил.

4. Новиков Б. Настройка приложений баз данных / Борис Новиков, Генриетта Домбровская. - СПб.: БХВ-Петербург, 2012. - 240 с.: ил.

5. Селко, Д. Стиль программирования Джо Селко на SQL: пер. с англ. / Джо Селко. - М.: Русская редакция; СПб.: Питер, 2006. - 206 с.: ил.

6. Официальный сайт PostgreSQL: http://www.postgresql.org

7. Postgres Professional: http://postgrespro.ru

Д.М.Лысанов., А.Г.Исавнин

Системы управления базами данных.

Учебно-методическое пособие

Подписано в печать 22.09.2023. Формат 60x84/16.
Печать ризографическая. Бумага офсетная.
Гарнитура «Times New Roman».
Усл.п.л. 5.3125 Уч.-изд. л. 5.1
Тираж 100 экз. Заказ № 1235

Отпечатано в Издательско-полиграфическом центре
Набережночелнинского института
Казанского (Приволжского) федерального университета

423810, г. Набережные Челны, Новый город, пр.Мира, 68/19
тел./факс (8552) 39-65-99 e-mail: ic-nchi-kpfu@mail.ru

ЛЫСАНОВ Д.М, ИСАВНИН А.Г