PostgresPro

Язык SQL

Лекция 3
Основы языка определения данных

Е. П. Моргунов
Сибирский государственный университет науки и технологий
имени академика М. Ф. Решетнева
г. Красноярск
Институт информатики и телекоммуникаций
emorgunov@mail.ru

Компания Postgres Professional
г. Москва

Версия от 28.08.2019

Подготовка к работе

PostgresPro

На вашем компьютере уже должна быть развернута база данных demo.

- Войдите в систему как пользователь postgres:

```
su - postgres
```

- Должен быть запущен сервер баз данных PostgreSQL.

```
pg_ctl  start  -D /usr/local/pgsql/data  -l postgres.log
```

- Для проверки запуска сервера выполните команду

```
pg_ctl  status  -D /usr/local/pgsql/data
```

- или

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

- Запустите утилиту psql и подключитесь к базе данных demo

psql -d demo -U postgres (можно просто **psql -d demo**)

- Для останова сервера баз данных PostgreSQL служит команда

```
pg_ctl  stop  -D /usr/local/pgsql/data  -l postgres.log
```

Вводные сведения

PostgresPro

- Язык SQL традиционно разделяется на две группы команд.
- Первая из них предназначена для определения данных, т. е. для создания объектов базы данных, таких, например, как таблицы.
- Вторая группа команд служит для выполнения различных операций с данными, таких, как вставка строк в таблицы, выполнение запросов к ним, обновление и удаление строк из таблиц.
- В этой лекции мы сосредоточимся на командах первой группы, т. е. на *определении данных*.
- Рассмотрим все таблицы базы данных «Авиаперевозки».

3.1. Значения по умолчанию и ограничения целостности

Небольшое замечание

PostgresPro

Основные сведения о значениях по умолчанию и ограничениях мы проиллюстрируем на той простой базе данных, состоящей из двух таблиц — «Студенты» и «Успеваемость», о которой речь шла также в первой главе пособия.

Таблица «Студенты»

PostgresPro

Описание атрибута	Имя атрибута	Тип данных	Тип PostgreSQL	Ограничения
No зачетной книжки	record_book	Числовой	numeric(5)	NOT NULL
Ф. И. О.	name	Символьный	text	NOT NULL
Серия документа	doc_ser	Числовой	numeric(4)	
Номер документа	doc_num	Числовой	numeric(6)	

По мере рассмотрения ограничений будет становиться понятно назначение каждого из них в обеих таблицах.

Таблица «Успеваемость»

PostgresPro

Описание атрибута	Имя атрибута	Тип данных	Тип PostgreSQL	Ограничения
No зачетной книжки	record_book	Числовой	numeric(5)	NOT NULL
Учебная дисциплина	subject	Символьный	text	NOT NULL
Учебный год	acad_year	Символьный	text	NOT NULL
Семестр	term	Числовой	numeric(1)	NOT NULL term = 1 OR term = 2
Оценка	mark	Числовой	numeric(1)	DEFAULT 5 mark >= 3 AND mark <= 5

Значения по умолчанию

PostgresPro

- При работе с базами данных нередко возникают ситуации, когда то или иное значение является *типичным* для какого-то конкретного столбца.
- Например, если мы при проектировании таблицы «Успеваемость» (progress) знаем, что успехи студентов, как правило, заслуживают оценки «отлично», то в команде CREATE TABLE мы можем отразить этот факт с помощью ключевого слова DEFAULT:

```
CREATE TABLE progress
...
  mark numeric( 1 ) DEFAULT 5,
...
```

Ограничение CHECK (1)

PostgresPro

- Это ограничение бывает двух видов: ограничение уровня *атрибута* и уровня *таблицы*.
- Различие между ними только в синтаксическом оформлении: в обоих случаях в выражении могут содержаться обращения не только к одному, но также и к нескольким атрибутам таблицы.
- В первом случае ограничение CHECK является частью определения одного конкретного атрибута, а во втором случае оно записывается как самостоятельный элемент определения таблицы.

```
CREATE TABLE progress
( ...
  term numeric( 1 ) CHECK ( term = 1 OR term = 2 ),
  mark numeric( 1 ) CHECK ( mark >= 3 AND mark <= 5 ),
  ...
);
```

Ограничение CHECK (2)

Каждое ограничение имеет имя. Мы можем задать его сами с помощью ключевого слова CONSTRAINT. Если же мы этого не сделаем, тогда СУБД сформирует имя автоматически.

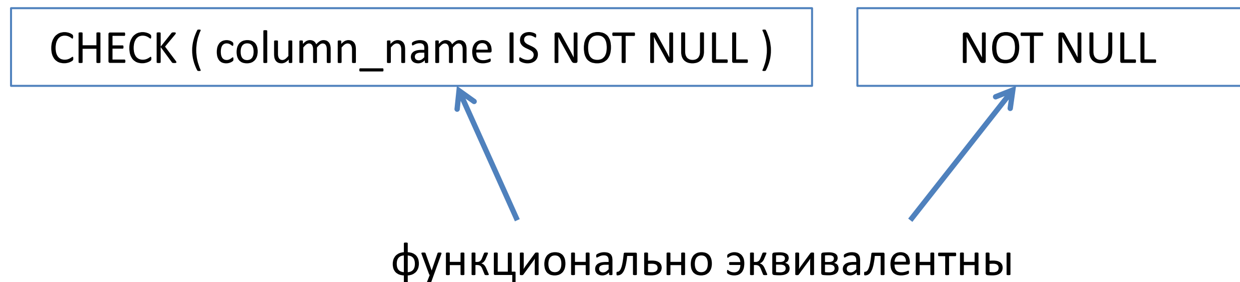
А вот ограничение уровня таблицы:

```
CREATE TABLE progress
( ...
  mark numeric( 1 ),
  CONSTRAINT valid_mark CHECK ( mark >= 3 AND mark <= 5 ),
  ...
);
```

Ограничение NOT NULL

PostgresPro

- Оно означает, что в столбце таблицы, на который наложено это ограничение, должны обязательно присутствовать какие-либо *определенные* значения.
- При разработке баз данных, исходя из логики конкретной предметной области, зачастую требуется использовать это ограничение.
- Как сказано в документации, оно функционально эквивалентно ограничению CHECK (column_name IS NOT NULL), но в PostgreSQL создание явного ограничения NOT NULL является более эффективным подходом.



Ограничение уникальности UNIQUE (1)

PostgresPro

- Такое ограничение, наложенное на конкретный столбец, означает, что все значения, содержащиеся в этом столбце в различных строках таблицы, должны быть *уникальными*, т. е. не должны повторяться.
- Ограничение уникальности может включать в себя и *несколько* столбцов. В этом случае уникальной должна быть уже *комбинация их значений*.
- Когда в ограничение уникальности включается только один столбец, то можно задать ограничение непосредственно в определении столбца.

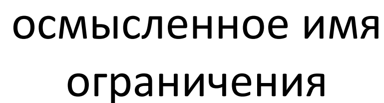
Ограничение уникальности UNIQUE (2)

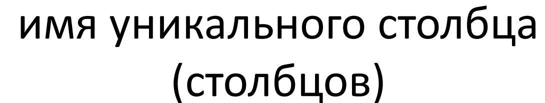
PostgresPro

```
CREATE TABLE students
( record_book numeric( 5 ) UNIQUE,
  ...
```

Это ограничение можно было бы записать и так, дав ему осмысленное имя:

```
CREATE TABLE students
( record_book numeric( 5 ),
  name text NOT NULL,
  ...
  CONSTRAINT unique_record_book UNIQUE ( record_book ),
  ...
);
```





Ограничение уникальности UNIQUE (3)

PostgresPro

Опять обратимся к таблице «Студенты» (students) и покажем, как можно создать ограничение уникальности, включающее более одного столбца.

```
CREATE TABLE students
( ...
  doc_ser numeric( 4 ),
  doc_num numeric( 6 ),
  ...
  CONSTRAINT unique_passport UNIQUE ( doc_ser, doc_num ),
  ...
);
```

ВАЖНО! При добавлении ограничения уникальности автоматически создается индекс на основе B-дерева для поддержки этого ограничения.

Первичный ключ – немного теории

PostgresPro

- Этот ключ является уникальным идентификатором строк в таблице.
- Ключ может быть как простым, т. е. включать только один атрибут, так и составным, т. е. включать более одного атрибута.
- При этом в отличие от уникального ключа, определяемого с помощью ограничения UNIQUE, *атрибуты, входящие в состав первичного ключа, не могут иметь значений NULL*.
- Таким образом, определение первичного ключа эквивалентно определению уникального ключа, дополненного ограничением NOT NULL.
- Однако не стоит в реальной работе заменять первичный ключ комбинацией ограничений UNIQUE и NOT NULL, поскольку теория баз данных требует наличия в каждой таблице именно первичного ключа.
- Первичный ключ является частью метаданных, его наличие позволяет другим таблицам использовать его в качестве уникального идентификатора строк в данной таблице.
- Это удобно, например, при создании внешних ключей, речь о которых пойдет ниже.
- Перечисленными свойствами обладает также и уникальный ключ.

Первичный ключ – примеры (1)

PostgresPro

Если первичный ключ состоит из одного атрибута, то можно указать его непосредственно в определении этого атрибута:

```
CREATE TABLE students
( record_book numeric( 5 ) PRIMARY KEY,
  ...
);
```

А можно сделать это и в виде отдельного ограничения:

```
CREATE TABLE students
( record_book numeric( 5 ),
  ...
  PRIMARY KEY ( record_book )
);
```

Первичный ключ – примеры (2)

PostgresPro

- В случае создания составного первичного ключа имена столбцов, входящих в его состав, перечисляются в выражении PRIMARY KEY через запятую:

```
PRIMARY KEY ( column1, column2, ...)
```

- При добавлении первичного ключа автоматически создается *индекс* на основе *B-дерева* для поддержки этого ограничения.
- В таблице может быть любое число ограничений UNIQUE, дополненных ограничением NOT NULL, но первичный ключ может быть только один.
- PostgreSQL допускает и отсутствие первичного ключа, хотя строгая теория реляционных баз данных *не рекомендует так поступать*.

Внешний ключ – немного теории (1)

PostgresPro

- Внешние ключи являются средством поддержания так называемой **ссылочной целостности** (referential integrity) между связанными таблицами.
- Напомним, что это означает, на примере таблиц «Студенты» (students) и «Успеваемость» (progress). В первой из них содержатся данные о студентах, а во второй — сведения об их успеваемости. Поскольку в процессе обучения студенты сдают целый ряд зачетов и экзаменов, то в таблице «Успеваемость» для каждого студента может присутствовать несколько строк.
- Для большинства из них это так и будет, хотя, в принципе, возможна ситуация, когда для какого-то студента в таблице «Успеваемость» не окажется ни одной строки (если, он, например, находится в академическом отпуске или был отчислен, не сдав ни одного экзамена).

Внешний ключ – немного теории (2)

PostgresPro

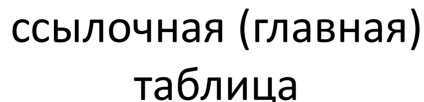
- Конечно, должна быть возможность определить, какому студенту принадлежат те или иные оценки, т. е. какие строки в таблице «Успеваемость» с какими строками в таблице «Студенты» связаны.
- Для решения этой задачи не требуется в каждой строке таблицы «Успеваемость» повторять все сведения о студенте: номер зачетной книжки, фамилию, имя и отчество, данные документа, удостоверяющего личность.
- Достаточно включить в состав каждой строки таблицы «Успеваемость» лишь уникальный идентификатор строки из таблицы «Студенты». В нашем случае это будет номер зачетной книжки — record_book. Данный атрибут и будет являться *внешним ключом* таблицы «Успеваемость».
- Таким образом, получив строку из таблицы «Студенты», можно будет найти все соответствующие ей строки в таблице «Успеваемость», сопоставив значения атрибутов record_book в строках обеих таблиц. В результате мы сможем получить все строки таблицы «Успеваемость», связанные с конкретной строкой из таблицы «Студенты» по внешнему ключу.

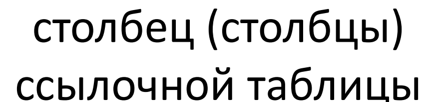
Внешний ключ – немного теории (3)

PostgresPro

- Таблица «Успеваемость» будет **ссылающейся** (referencing), а таблица «Студенты» — **ссылочной** (referenced).
- Обратите внимание, что *внешний ключ* ссылающейся таблицы ссылается на *первичный ключ* ссылочной таблицы. Допускается ссылка и на уникальный ключ, не являющийся первичным.
- В данном контексте для описания отношений между таблицами можно сказать, что таблица students является *главной*, а таблица progress — *подчиненной*.
- Создать внешний ключ можно в формате ограничения уровня атрибута следующим образом:

```
CREATE TABLE progress
( record_book numeric( 5 )
    REFERENCES students ( record_book ),
...
);
```





Внешний ключ – немного теории (4)

PostgresPro

- Предложение REFERENCES создает ограничение ссылочной целостности и указывает в качестве ссылочного ключа атрибут record_book.
- Это означает, что в таблицу «Успеваемость» (progress) нельзя ввести строку, значение атрибута record_book которой отсутствует в таблице «Студенты» (students). Говоря простым языком, нельзя ввести запись об оценке того студента, информация о котором еще не введена в таблицу «Студенты».
- Поскольку внешний ключ в нашем примере ссылается на первичный ключ, можно использовать сокращенную форму записи этого ограничения, не указывая список атрибутов:

```
CREATE TABLE progress
( record_book numeric( 5 ) REFERENCES students,
  ...
);
```

Внешний ключ – примеры

PostgresPro

- Можно определить внешний ключ и в форме ограничения уровня таблицы:

```
CREATE TABLE progress
( record_book numeric( 5 ),
  ...
  FOREIGN KEY ( record_book )
    REFERENCES students ( record_book )
);
```

- ВАЖНО! Число атрибутов и их типы данных (и *домены*!) во внешнем ключе ссылающейся таблицы и в первичном ключе ссылочной таблицы должны быть согласованы.
- Ограничению внешнего ключа можно присвоить наименование, как и любому другому ограничению, с помощью ключевого слова CONSTRAINT.

Обеспечение ссылочной целостности (1)

PostgresPro

- При наличии связей между таблицами, организованных с помощью внешних ключей, необходимо придерживаться *определенной политики* при выполнении операций удаления и обновления строк в ссылочных таблицах, т. е. в тех, *на которые* ссылаются другие таблицы.
- В нашем примере ситуация принятия «политического» решения возникает при удалении строк из таблицы «Студенты» (students).
- Тогда возникает закономерный вопрос: что делать со строками в таблице «Успеваемость» (progress), которые ссылаются на удаляемую строку в таблице «Студенты» (students)?
- Возможны несколько вариантов.

Обеспечение ссылочной целостности (2)

PostgresPro

1. *Удаление связанных строк* из таблицы «Успеваемость» (progress).
2. *Запрет удаления строки* из таблицы «Студенты» (students), если в таблице «Успеваемость» (progress) есть хотя бы одна строка, ссылающаяся на удаляемую строку в таблице «Студенты»
3. *Присваивание* атрибутам внешнего ключа в строках таблицы «Успеваемость» *значения NULL*. (Если нет ограничения NOT NULL.)
4. *Присваивание* атрибутам внешнего ключа в строках таблицы «Успеваемость» (progress) *значения DEFAULT*, если оно, конечно, было предписано при создании таблицы.
   Пример. При удалении какого-то отдела в большой организации его сотрудники временно переподчиняются другому отделу, код которого как раз и указывается в предложении DEFAULT.

Удаление связанных строк

PostgresPro

- Удаление связанных строк из таблицы «Успеваемость» (progress), что означает, что при отчислении студента будет удаляться вся история его успехов в учебе.
- Эта операция называется **каскадным удалением** и для ее реализации в определение внешнего ключа добавляются ключевые слова ON DELETE CASCADE. Например:

```
CREATE TABLE progress
( record_book numeric( 5 ),
  ...
  FOREIGN KEY ( record_book )
    REFERENCES students ( record_book )
    ON DELETE CASCADE
);
```

Запрет удаления строки из главной таблицы (1)

PostgresPro

- Запрет удаления строки из таблицы «Студенты» (students), если в таблице «Успеваемость» (progress) есть хотя бы одна строка, ссылающаяся на удаляемую строку в таблице «Студенты».
- Для реализации такой политики в определение внешнего ключа добавляются ключевые слова ON DELETE RESTRICT или ON DELETE NO ACTION. Если в определении внешнего ключа не предписано конкретное действие, то по умолчанию используется NO ACTION.
- Оба эти варианта означают, что если в ссылающейся таблице, т. е. «Успеваемость», есть строки, ссылающиеся на удаляемую строку в таблице «Студенты», то операция удаления будет отменена, и будет выведено сообщение об ошибке.
- Отличие между этими двумя вариантами лишь в том, что при использовании NO ACTION *можно отложить проверку* выполнения ограничения на более поздний строк в рамках транзакции, а в случае RESTRICT проверка выполняется немедленно.

Запрет удаления строки из главной таблицы (2)

Поэтому если бы внешний ключ определили таким образом:

```
CREATE TABLE progress
( record_book numeric( 5 ),
  ...
  FOREIGN KEY ( record_book )
    REFERENCES students ( record_book ) ON DELETE RESTRICT
);
```

или таким:

```
CREATE TABLE progress
( record_book numeric( 5 ),
  ...
  -- по умолчанию NO ACTION
  FOREIGN KEY ( record_book )
    REFERENCES students ( record_book )
);
```

то при попытке удаления строки из таблицы «Студенты» и наличии в таблице «Успеваемость» строк, связанных с ней, операция удаления была бы отменена с выводом сообщения об ошибке.

Присваивание значения NULL

PostgresPro

- Присваивание атрибутам внешнего ключа в строках таблицы «Успеваемость» значения NULL.
- Для реализации этого подхода необходимо, чтобы на атрибуты внешнего ключа не было наложено ограничение NOT NULL.
- Оформляется этот вариант так:

```
CREATE TABLE progress
( record_book numeric( 5 ),
  ...
  FOREIGN KEY ( record_book )
    REFERENCES students ( record_book )
    ON DELETE SET NULL
);
```

Присваивание значения DEFAULT

PostgresPro

- Присваивание атрибутам внешнего ключа в строках таблицы «Успеваемость»
- (progress) значения DEFAULT, если оно, конечно, было предписано при создании таблицы.
- Оформляется этот вариант так (значение во фразе DEFAULT взято произвольное):

```
CREATE TABLE progress
( record_book numeric( 5 ) DEFAULT 12345,
  ...
  FOREIGN KEY ( record_book )
    REFERENCES students ( record_book )
    ON DELETE SET DEFAULT
);
```

- Важно учитывать, что если в ссылочной таблице нет строки *с тем же значением* ключевого атрибута, которое было предписано во фразе DEFAULT при создании ссылающейся таблицы, то будет иметь место нарушение ограничения ссылочной целостности и операция удаления не будет выполнена.

Что происходит при обновлении строк?

PostgresPro

- При выполнении операции UPDATE используются эти же варианты подходов по отношению к обеспечению ссылочной целостности.
- Аналогом каскадного удаления является каскадное обновление:

```
CREATE TABLE progress
( record_book numeric( 5 ),
  ...
  FOREIGN KEY ( record_book )
    REFERENCES students ( record_book )
    ON UPDATE CASCADE
);
```

- В этом случае измененные значения ссылочных атрибутов копируются в ссылающиеся строки ссылающейся таблицы, т. е. новое значение атрибута record_book из строки таблицы «Студенты» будет скопировано во все строки таблицы «Успеваемость», ссылающиеся на обновленную строку.

Создадим таблицу «Студенты»

PostgresPro

Прежде чем создавать таблицы, создайте базу данных edu из среды операционной системы:

```
createdb -U postgres edu
```

Подключитесь к ней:

```
psql -d edu -U postgres
```

```
CREATE TABLE students
( record_book numeric( 5 ) NOT NULL,
  name text NOT NULL,
  doc_ser numeric( 4 ),
  doc_num numeric( 6 ),
  PRIMARY KEY ( record_book )
);
```

Создадим таблицу «Успеваемость»

PostgresPro

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

3.2. Создание и удаление таблиц

Подготовка к работе

PostgresPro

Теперь запустите утилиту psql и подключитесь к базе данных demo с учетной записью пользователя postgres:

```
psql -d demo -U postgres
```

Выберите в качестве текущей схемы схему bookings:

```
SET search_path TO bookings;
```

или

```
SET search_path = bookings;
```

Для подключения к базе данных demo изнутри psql сделайте так:

```
\connect demo
```

Существует и сокращенный вариант этой команды:

```
\c demo
```

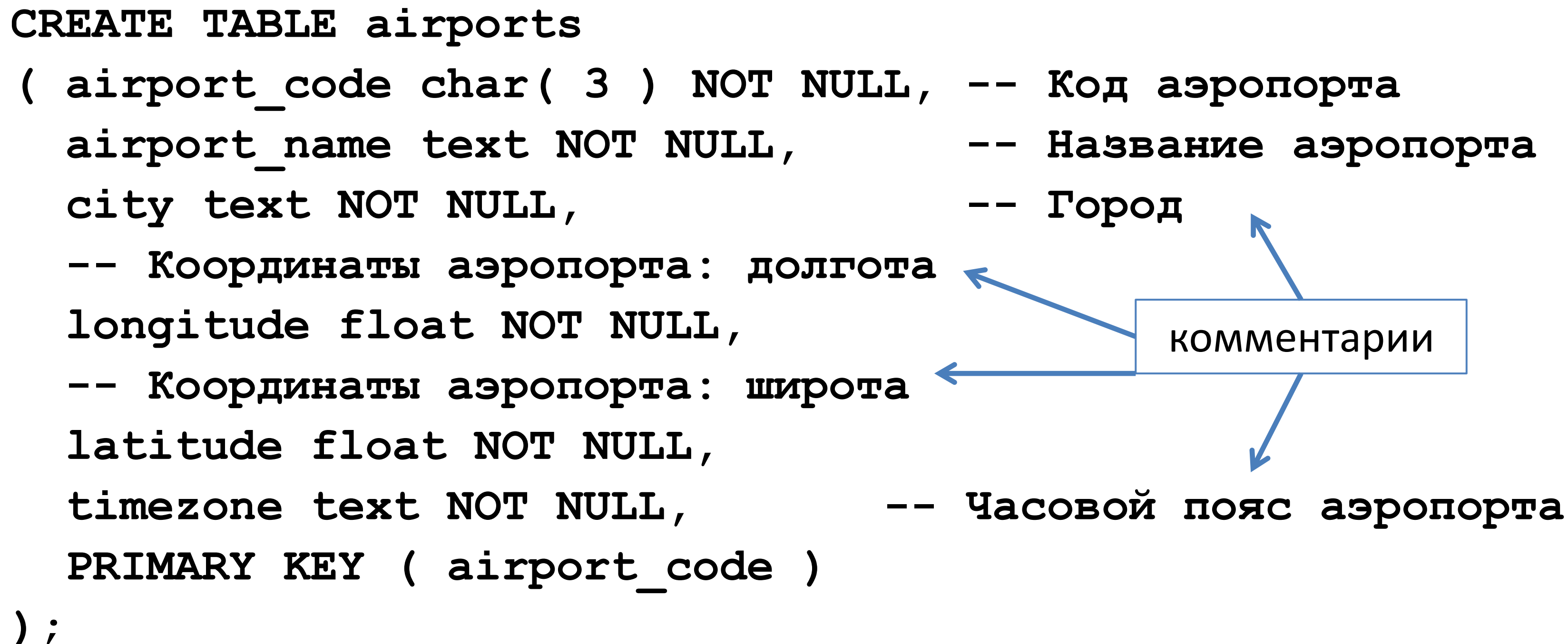
Ряд замечаний

PostgresPro

- При создании таблиц необходимо учитывать связи между ними.
- Поэтому сначала должны создаваться ссылочные таблицы, а потом — ссылающиеся.
- При наличии циклических ссылок таблиц друг на друга придется воспользоваться командой ALTER TABLE, о которой речь пойдет в следующем разделе этой лекции.

ВАЖНО! В схеме bookings все таблицы уже созданы. Выполнять команды CREATE TABLE уже не нужно.

Таблица «Аэропорты»




```
CREATE TABLE airports
( airport_code char( 3 ) NOT NULL, -- Код аэропорта
  airport_name text NOT NULL,      -- Название аэропорта
  city text NOT NULL,              -- Город
  -- Координаты аэропорта: долгота
  longitude float NOT NULL,
  -- Координаты аэропорта: широта
  latitude float NOT NULL,
  timezone text NOT NULL,       -- Часовой пояс аэропорта
  PRIMARY KEY ( airport_code )
);
```

Что получилось? (1)

PostgresPro

\d airports

или

\d airp

а затем нажать клавишу Tab (имя таблицы будет дополнено)

```
Таблица "bookings.airports"
    Столбец    |         Тип          | Модификаторы
--------------+--------------------+-------------
 airport_code | character(3)         | NOT NULL
 airport_name | text                 | NOT NULL
 city         | text                 | NOT NULL
 longitude    | double precision | NOT NULL
 latitude     | double precision | NOT NULL
 timezone     | text                 | NOT NULL
```

Что получилось? (2)

```
Индексы:
    "airports_pkey" PRIMARY KEY, btree (airport_code)
Ссылки извне:
    TABLE "flights" CONSTRAINT
"flights_arrival_airport_fkey"
      FOREIGN KEY (arrival_airport)
      REFERENCES airports(airport_code)
    TABLE "flights" CONSTRAINT
"flights_departure_airport_fkey"
      FOREIGN KEY (departure_airport)
      REFERENCES airports(airport_code)
```

Команда COMMENT

PostgresPro

- PostgreSQL предлагает свое расширение — команду COMMENT, которая позволяет создавать комментарии (описания) к различным объектам базы данных.
- Эти комментарии будут также сохраняться в базе данных.
- Например, для создания описания столбца city таблицы airports нужно сделать так:

```
COMMENT ON COLUMN airports.city IS 'Город';
```

- Чтобы увидеть описания столбцов таблицы, нужно в команде \d добавить символ «+», например:

```
\d+ airports
```

Таблица «Рейсы» (1)

PostgresPro

```
CREATE TABLE flights
( flight_id  serial NOT NULL,      -- Идентификатор рейса
  flight_no char( 6 ) NOT NULL,    -- Номер рейса
  -- Время вылета  по расписанию
  scheduled_departure timestamptz NOT NULL,
  -- Время прилета  по расписанию
  scheduled_arrival timestamptz NOT NULL,
  -- Аэропорт отправления
  departure_airport char( 3 ) NOT NULL,
  arrival_airport char( 3 ) NOT NULL, -- Аэропорт прибытия
  status varchar( 20 ) NOT NULL,        -- Статус рейса
  aircraft_code char( 3 ) NOT NULL,  -- Код самолета, IATA
  -- Фактическое время вылета
  actual_departure timestamptz,
  -- Фактическое время прилета
  actual_arrival  timestamptz,
  ...
```

Таблица «Рейсы» (2)

PostgresPro

```
CREATE TABLE flights
(
  ...
  CHECK ( scheduled_arrival > scheduled_departure ),
  CHECK ( actual_arrival IS NULL OR
          ( actual_departure IS NOT NULL AND
            actual_arrival IS NOT NULL AND
            actual_arrival > actual_departure
          )
        ),
  CHECK ( status IN ( 'On Time', 'Delayed', 'Departed',
                      'Arrived', 'Scheduled',
                      'Cancelled' ) ),
  ...
);
```

Это условие лишнее?

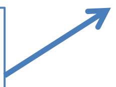


Таблица «Рейсы» (3)

PostgresPro

```
CREATE TABLE flights
(
  ...
  PRIMARY KEY ( flight_id ),
  UNIQUE ( flight_no, scheduled_departure ),
  FOREIGN KEY ( aircraft_code )
    REFERENCES aircrafts ( aircraft_code ),
  FOREIGN KEY ( arrival_airport )
    REFERENCES airports ( airport_code ),
  FOREIGN KEY ( departure_airport )
    REFERENCES airports ( airport_code )
);
```

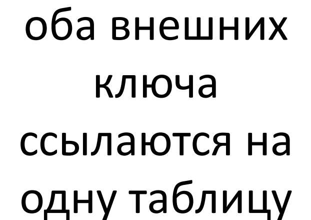


Таблица «Рейсы» – комментарии (1)

PostgresPro

- В качестве первичного ключа используется так называемый суррогатный ключ, состоящий из одного атрибута — flight_id. Тип данных этого атрибута — serial, т. е. значения целого типа для этого атрибута будут извлекаться из последовательности.
- **Суррогатный ключ** — это уникальный ключ, назначение которого — только идентифицировать строки в таблице.
- Зачастую для него используются целочисленные значения. Такому ключу не соответствует *никакое свойство никакой сущности реального мира*.
- Это — абстракция, позволяющая в ряде случаев упростить определения таблиц, например, за счет сокращения числа атрибутов во внешних ключах до одного.
- В нашей таблице «Рейсы» (flights) суррогатный ключ как раз и служит для того, чтобы в таблицах, ссылающихся на нее, внешние ключи состояли только из атрибута flight_id.

Таблица «Рейсы» – комментарии (2)

PostgresPro

- Конечно, существует и *естественный уникальный ключ*, состоящий из двух атрибутов: номер рейса (flight_no) и время вылета по расписанию (scheduled_departure).
- Для него нам придется создать уникальный ключ, чтобы избежать дублирования значений: очевидно, что в один и тот же момент времени не могут выполняться два (и более) рейса, имеющие один и тот же номер.
- Для атрибутов, имеющих смысл даты/времени, выбран тип данных timestamptz, т. е. временная отметка с указанием часового пояса.
- Это важно, т. к. перелеты могут совершаться между городами, находящимися в разных часовых поясах, а время вылета и время прилета указывается местное.
- Подумайте, нельзя ли упростить выражение для второго ограничения CHECK.

Что получилось? (1)

PostgresPro

```
\d flights
```

- Обратите внимание, что для атрибута flight_id указан тип данных integer, а не serial, как предписано в команде для создания этой таблицы.
- О том, что значения для этого столбца будут формироваться с помощью последовательности, говорит фраза

```
DEFAULT nextval('flights_flight_id_seq'::regclass)
```

- В этой фразе указано и имя последовательности — flights_flight_id_seq.

Что получилось? (2)

PostgresPro

Если выполнить команду

\d

то можно увидеть эту последовательность в списке объектов базы данных.

```
Список отношений
   Схема   |            Имя            |          Тип          |
Владелец
----------+------------------------+---------------------+--
--------
  ...
 bookings | flights_flight_id_seq | последовательность |
postgres
  ...
(11 строк)
```

Чтобы посмотреть описание последовательности flights_flight_id_seq, нужно использовать команду \d:

\d flights_flight_id_seq

Таблица «Бронирования»

```
CREATE TABLE bookings
( book_ref  char( 6 ) NOT NULL,    -- Номер бронирования
  book_date timestamptz NOT NULL, -- Дата бронирования
  -- Полная  стоимость бронирования
  total_amount numeric( 10, 2 ) NOT NULL,
  PRIMARY KEY ( book_ref )
);
```

Для атрибута «Дата бронирования» (book_date) выбран тип данных timestamptz — временная отметка с часовым поясом, т. к. билеты могут приобретаться в городах, находящихся в различных часовых поясах.

В случаях, требующих точных вычислений, необходимо использовать числа с фиксированной точностью. Поэтому для атрибута «Полная сумма бронирования» (total_amount) выбирается тип данных numeric, при этом масштаб, т. е. число цифр справа от десятичной точки (запятой), будет равен 2.

```
\d bookings
```

Вы увидите в том числе и ссылки извне, т. е. те таблицы, которые ссылаются на эту таблицу.

Таблица «Билеты»

PostgresPro

```
CREATE TABLE tickets
( ticket_no char( 13 ) NOT NULL, -- Номер билета
  book_ref  char( 6 ) NOT NULL,  -- Номер бронирования
  -- Идентификатор пассажира
  passenger_id varchar( 20 ) NOT NULL,
  passenger_name text NOT NULL,  -- Имя пассажира
  contact_data jsonb,       -- Контактные данные пассажира
  PRIMARY KEY ( ticket_no ),
  FOREIGN KEY ( book_ref )
    REFERENCES bookings ( book_ref )
);
```

- Хотя уникальные тринадцатизначные номера билетов — числовые, но в них могут присутствовать лидирующие нули, поэтому числовой тип данных здесь не годится, а приходится использовать тип character (сокращенно — char).
- Очень интересный атрибут «Контактные данные пассажира» (contact_data). Его особенность в том, что эти данные могут иметь некоторую структуру, но при этом создавать дополнительные атрибуты в таблице нецелесообразно. С такими данными — их называют полуструктурированными — PostgreSQL хорошо умеет работать. Его тип **jsonb**.

```
\d tickets
```

Таблица «Перелеты» (1)

PostgresPro

```
CREATE TABLE ticket_flights
( ticket_no char( 13 ) NOT NULL,     -- Номер билета
  flight_id integer NOT NULL,        -- Идентификатор рейса
  -- Класс обслуживания
  fare_conditions varchar( 10 ) NOT NULL,
  amount numeric( 10, 2 ) NOT NULL, -- Стоимость перелета
  CHECK ( amount >= 0 ),
  CHECK ( fare_conditions IN ( 'Economy', 'Comfort',
                               'Business' ) ),
  PRIMARY KEY ( ticket_no, flight_id ),
  FOREIGN KEY ( flight_id )
    REFERENCES flights ( flight_id ),
  FOREIGN KEY ( ticket_no )
    REFERENCES tickets ( ticket_no )
);
```

Таблица «Перелеты» (2)

PostgresPro

- Перелеты вписываются в электронные билеты, при этом в каждый электронный билет может быть вписано более одного перелета. Поэтому первичным ключом будет комбинация двух атрибутов: «Номер билета» (ticket_no) и «Идентификатор рейса» (flight_id).
- Атрибут «Стоимость перелета» (amount) требует использования типа данных numeric, поскольку денежные суммы должны записываться с определенной точностью, а гарантировать ее может только тип данных numeric. Число цифр после запятой принимается равным двум.
- Оба атрибута, составляющих первичный ключ, в свою очередь, сами являются внешними ключами.

```
\d ticket_flights
```

Таблица «Посадочные талоны» (1)

PostgresPro

```
CREATE TABLE boarding_passes
( ticket_no  char( 13 ) NOT NULL, -- Номер билета
  flight_id integer NOT NULL,     -- Идентификатор рейса
  -- Номер посадочного талона
  boarding_no integer NOT NULL,
  seat_no varchar( 4 ) NOT NULL,  -- Номер места
  PRIMARY KEY ( ticket_no, flight_id ),
  UNIQUE ( flight_id, boarding_no ),
  UNIQUE ( flight_id, seat_no ),
  FOREIGN KEY ( ticket_no, flight_id )
    REFERENCES ticket_flights ( ticket_no, flight_id )
);


\d boarding_passes
```

Таблица «Посадочные талоны» (2)

PostgresPro

- Номер посадочного талона — это просто целое число, порядковый номер пассажира при регистрации билетов на конкретный рейс, поэтому тип данных выбирается integer.
- Обратите внимание, что эта таблица имеет связь с таблицей «Перелеты» (ticket_flights) типа 1:1. Это объясняется тем, что пассажир, купивший билет на конкретный рейс, при регистрации получает только один посадочный талон.
- Конечно, если пассажир на регистрацию не явился, он не получает талона. Поэтому число строк в таблице «Посадочные талоны» может в общем случае оказаться меньше числа строк в таблице «Перелеты».
- Логично ожидать, что первичные ключи у этих двух таблиц будут одинаковыми: они включают атрибуты «Номер билета» (ticket_no) и «Идентификатор рейса» (flight_id).

Таблица «Посадочные талоны» (3)

PostgresPro

- Поскольку таблица «Перелеты» все же является главной в этой связке таблиц, то в таблице «Посадочные талоны» создается внешний ключ, ссылающийся на нее. А поскольку тип связи между таблицами — 1:1, то внешний ключ совпадает с первичным ключом.
- Известно, что номер конкретного места в самолете пассажир получает при регистрации билета, а не при его бронировании, поэтому атрибут «Номер места» (seat_no) находится в таблице «Посадочные талоны», а не в таблице «Перелеты».
- Нельзя допустить, чтобы на одно место в салоне были направлены два и более пассажиров, поэтому создается уникальный ключ с атрибутами «Идентификатор рейса» (flight_id) и «Номер места» (seat_no).
- Еще один уникальный ключ призван гарантировать несовпадение номеров посадочных талонов на данном рейсе, он включает атрибуты «Идентификатор рейса» (flight_id) и «Номер посадочного талона» (boarding_no).

Связи данной таблицы

PostgresPro

- В процессе создания таблиц между ними образовывались связи за счет *внешних ключей*.
- Эти связи в описании таблицы можно увидеть, образно говоря, с двух сторон: таблицы, на которые ссылается данная таблица, указываются во фразе «Ограничения внешнего ключа», а таблицы, которые ссылаются на данную таблицу, указываются во фразе «Ссылки извне». Например:

```
\d tickets
...
Ограничения внешнего ключа:
    "tickets_book_ref_fkey" FOREIGN KEY (book_ref)
    REFERENCES bookings(book_ref)
Ссылки извне:
    TABLE "ticket_flights"
    CONSTRAINT "ticket_flights_ticket_no_fkey"
    FOREIGN KEY (ticket_no)
      REFERENCES tickets(ticket_no)
```

Удаление таблиц

PostgresPro

```
DROP TABLE aircrafts;
ОШИБКА: удалить объект таблица aircrafts нельзя, так как
от него зависят другие объекты
ПОДРОБНОСТИ: ограничение flights_aircraft_code_fkey в
отношении таблица flights зависит от объекта таблица
aircrafts
ограничение seats_aircraft_code_fkey в отношении таблица
 seats зависит от объекта таблица aircrafts
ПОДСКАЗКА: Для удаления зависимых объектов используйте
DROP ... CASCADE.
```

Дело в том, что таблица «Самолеты» (aircrafts) является ссылочной для таблиц «Рейсы» (flights) и «Места» (seats), что и отражено в сообщении.

Выполнив команду

```
\d flights
```

мы увидим внешний ключ, ссылающийся на таблицу «Самолеты» (aircrafts).

Каскадное удаление таблиц

PostgresPro

DROP TABLE aircrafts CASCADE;

Теперь удаление таблицы прошло успешно, при этом из таблиц «Рейсы» (flights) и «Места» (seats) были удалены внешние ключи, ссылающиеся на удаленную таблицу aircrafts. Вот это сообщение:

```
ЗАМЕЧАНИЕ: удаление распространяется на еще 2 объекта
ПОДРОБНОСТИ: удаление распространяется на объект
ограничение flights_aircraft_code_fkey в отношении таблица
flights
удаление распространяется на объект ограничение
seats_aircraft_code_fkey в отношении таблица seats
DROP TABLE
```

Теперь внешних ключей, ссылающихся на таблицу aircrafts в таблицах flights и seats нет. Можно проверить это с помощью команд

\d flights

\d seats

Избежание ненужных сообщений

PostgresPro

А что если выполнить команду для удаления той же самой таблицы повторно?

```
DROP TABLE aircrafts CASCADE;
```

Ничего непоправимого не случится, просто СУБД выдаст сообщение об ошибке:

```
ОШИБКА: таблица "aircrafts" не существует
```

Однако бывают ситуации, когда заранее известно, что возможна попытка удаления несуществующей таблицы.

```
DROP TABLE IF EXISTS aircrafts CASCADE;
```

При использовании этой фразы в случае наличия интересующей нас таблицы выполняется ее удаление, в случае же ее отсутствия выводится замечание, а не ошибка, а также сообщение об успешном выполнении команды удаления таблицы:

```
ЗАМЕЧАНИЕ: таблица "aircrafts" не существует, пропускается
DROP TABLE
```

3.3. Модификация таблиц

Добавление столбца и ограничения

PostgresPro

Предположим, что нам понадобилось иметь в базе данных сведения о крейсерской скорости полета всех моделей самолетов, которые эксплуатируются в нашей авиакомпании. Следовательно, необходимо добавить столбец в таблицу «Самолеты» (aircrafts).

```
ALTER TABLE aircrafts
  ADD COLUMN speed integer NOT NULL CHECK( speed >= 300 );
ОШИБКА: столбец "speed" содержит значения NULL
```

Решение:

```
ALTER TABLE aircrafts ADD COLUMN speed integer;
UPDATE aircrafts SET speed = 807
  WHERE aircraft_code = '733';
UPDATE aircrafts SET speed = 851
  WHERE aircraft_code = '763';
...
ALTER TABLE aircrafts ALTER COLUMN speed SET NOT NULL;
ALTER TABLE aircrafts ADD CHECK( speed >= 300 );
\d aircrafts
```

Удаление ограничения и столбца

PostgresPro

Конечно, если необходимость наличия того или иного ограничения отпадет, его можно удалить:

```
ALTER TABLE aircrafts ALTER COLUMN speed DROP NOT NULL;
ALTER TABLE aircrafts
  DROP CONSTRAINT aircrafts_speed_check;
```

Обратите внимание, что для удаления ограничения CHECK нужно указать его имя, которое можно выяснить с помощью команды

```
\d aircrafts
```

Если мы решим не усложнять нашу базу данных дополнительной информацией, то можем удалить и столбец. Конечно, вовсе не обязательно предварительно удалять ограничения, наложенные на этот столбец.

```
ALTER TABLE aircrafts DROP COLUMN speed;
```

Изменение типа данных столбца (1)

PostgresPro

Давайте изменим тип данных для атрибутов «Координаты аэропорта: долгота» (longitude) и «Координаты аэропорта: широта» (latitude) с float (double precision) на numeric(5, 2).

Команда ALTER TABLE поддерживает и выполнение *более одного действия* за один раз.

```
SELECT * FROM airports;
ALTER TABLE airports
  ALTER COLUMN longitude SET DATA TYPE numeric( 5,2 ),
  ALTER COLUMN latitude SET DATA TYPE numeric( 5,2 );
SELECT * FROM airports;
```

ВАЖНО! В том случае, когда один тип данных изменяется на другой тип данных в пределах одной группы, например, оба типа — числовые, то проблем обычно не возникает.

Правдоподобная ситуация (1)

PostgresPro

- Однако если исходный и целевой типы данных относятся к разным группам, тогда потребуются некоторые дополнительные усилия.
- Предположим, что по результатам опытной эксплуатации базы данных «Авиаперевозки» мы пришли к выводу о том, что необходимо создать таблицу, содержащую коды и наименования классов обслуживания.

```
CREATE TABLE fare_conditions
( fare_conditions_code integer,
  fare_conditions_name varchar( 10 ) NOT NULL,
  PRIMARY KEY ( fare_conditions_code )
);
```

Добавим в новую таблицу необходимые данные:

```
INSERT INTO fare_conditions
VALUES ( 1, 'Economy' ),
       ( 2, 'Business' ),
       ( 3, 'Comfort' );
```

Правдоподобная ситуация (2)

PostgresPro

- Поскольку мы ввели в обращение числовые коды для классов обслуживания, то необходимо модифицировать определение таблицы «Места» (seats), а именно: тип данных столбца «Класс обслуживания» (fare_conditions) изменить с varchar(10) на integer.
- Для реализации такой задачи служит фраза USING команды ALTER TABLE. Однако такой вариант команды не сработает:

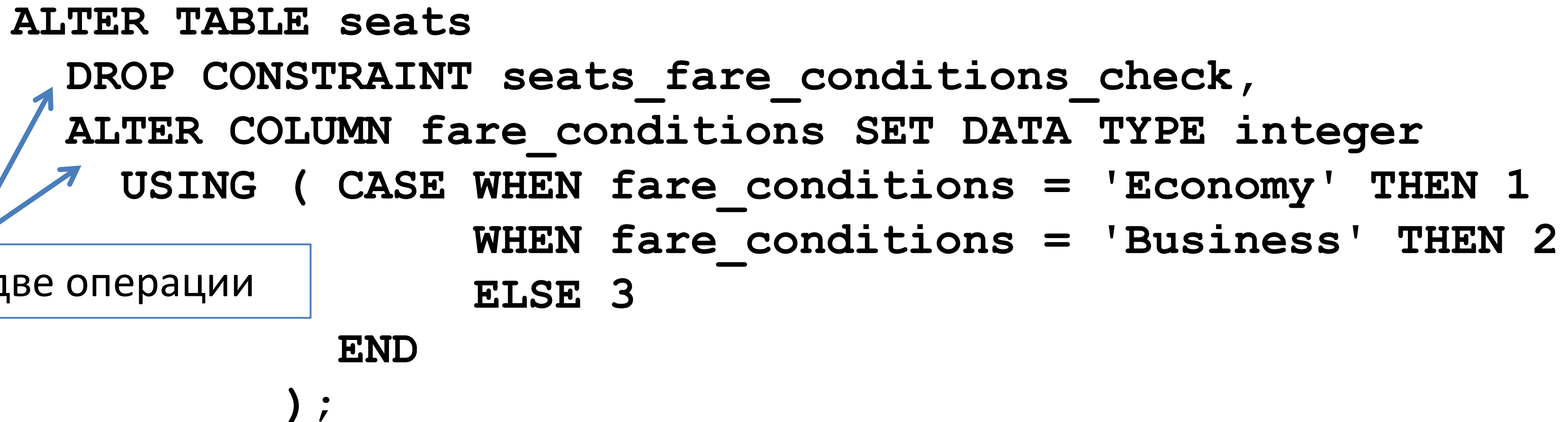
```
ALTER TABLE seats
  ALTER COLUMN fare_conditions SET DATA TYPE integer
    USING ( CASE WHEN fare_conditions = 'Economy' THEN 1
                 WHEN fare_conditions = 'Business' THEN 2
                 ELSE 3
            END
          );
ОШИБКА: ограничение-проверку "seats_fare_conditions_check"
нарушает некоторая строка
```

И в самом деле, в определении таблицы есть ограничение CHECK, которое требует, чтобы значение столбца fare_conditions выбиралось из списка: «Economy», «Comfort», «Business». При замене *символьных значений на числовые* это ограничение будет заведомо нарушаться.

Правдоподобная ситуация (3)

PostgresPro

В команду ALTER TABLE добавим операцию удаления этого ограничения:




```
ALTER TABLE seats
  DROP CONSTRAINT seats_fare_conditions_check,
  ALTER COLUMN fare_conditions SET DATA TYPE integer
    USING ( CASE WHEN fare_conditions = 'Economy' THEN 1
                 WHEN fare_conditions = 'Business' THEN 2
                 ELSE 3
            END
          );
```

Проверим результат работы с помощью команды

```
SELECT * FROM seats;
 aircraft_code | seat_no | fare_conditions
---------------+---------+-----------------
 319           | 2A      |               2
 319           | 2C      |               2
 319           | 2D      |               2
...
```

Правдоподобная ситуация (4)

PostgresPro

Теперь мы видим, что необходимо связать таблицы «Места» (seats) и «Классы обслуживания» (fare_conditions) по внешнему ключу. Сделаем это:

```
ALTER TABLE seats
  ADD FOREIGN KEY ( fare_conditions )
    REFERENCES fare_conditions ( fare_conditions_code );
```

Посмотрев описание таблицы «Места» (seats), увидим, что внешний ключ успешно создан.

```
\d seats
...
"seats_fare_conditions_fkey" FOREIGN KEY (fare_conditions)
REFERENCES fare_conditions(fare_conditions_code)
```

Правдоподобная ситуация (5)

PostgresPro

- Из теории известно, что атрибуты внешнего ключа *не обязательно должны ссылаться только на одноименные атрибуты* ссылочной таблицы.
- Однако для удобства сопровождения базы данных имеет смысл переименовать столбец fare_conditions в таблице «Места» (seats), т. е. дать ему имя fare_conditions_code:

```
ALTER TABLE seats
  RENAME COLUMN fare_conditions TO fare_conditions_code;
```

Имя атрибута, являющегося внешним ключом, изменилось, а вот имя ограничения seats_fare_conditions_fkey осталось неизменным:

```
"seats_fare_conditions_fkey" FOREIGN KEY
(fare_conditions_code)
 REFERENCES fare_conditions(fare_conditions_code)
```

Правдоподобная ситуация (6)

PostgresPro

Переименуем это ограничение, чтобы поддержать соблюдение правила именования ограничений:

```
ALTER TABLE seats
  RENAME CONSTRAINT seats_fare_conditions_fkey
    TO seats_fare_conditions_code_fkey;
```

Проверим, что получилось:

```
\d seats
```

Мы предусмотрели в таблице «Классы обслуживания» первичный ключ, но ведь значения атрибута «Наименование класса обслуживания» (fare_conditions_name) также должны быть уникальными, дублирование значений не допускается. Добавим ограничение уникальности по этому столбцу:

```
ALTER TABLE fare_conditions
  ADD UNIQUE ( fare_conditions_name );
```

Проверим, что получилось:

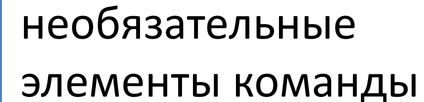
```
\d fare_conditions
```

3.4. Представления

Вводные сведения

- При работе с базами данных зачастую приходится многократно выполнять одни и те же запросы, которые могут быть весьма сложными и требовать обращения к нескольким таблицам. Чтобы избежать необходимости многократного формирования таких запросов, можно использовать так называемые **представления** (**views**).
- Если речь идет о выборке данных, то представления практически неотличимы от таблиц с точки зрения обращения к ним в командах SELECT.
- Упрощенный синтаксис команды CREATE VIEW, предназначенной для создания представлений, таков:

```
CREATE VIEW name [ ( column_name [, ...] ) ]
  AS query;
```



Если список имен столбцов не приведен, тогда их имена «вычисляются» (формируются) на основании текста запроса.

Простой пример

Задача: подсчитать количество мест в салонах для всех моделей самолетов с учетом класса обслуживания (бизнес-класс и экономический класс).

```
CREATE VIEW seats_by_fare_cond AS
  SELECT aircraft_code, fare_conditions, count( * )
  FROM seats
  GROUP BY aircraft_code, fare_conditions
  ORDER BY aircraft_code, fare_conditions;
```

Теперь мы можем вместо написания сложного первоначального запроса обращаться непосредственно к представлению, как будто это обычная таблица.

```
SELECT * FROM seats_by_fare_cond;
```

ВАЖНО! В отличие от таблиц, представления не содержат данных. Данные выбираются из таблиц, на основе которых представление создано, при каждом обращении к нему в команде SELECT.

Предложение OR REPLACE (1)

PostgresPro

- Предложение OR REPLACE – это *расширение* команды CREATE VIEW, которое предлагает PostgreSQL. Однако нужно помнить о том, что при создании новой версии представления (без явного удаления старой с помощью команды DROP VIEW) должны оставаться неизменными имена столбцов представления.
- Обратите внимание на добавление фразы OR REPLACE и ключевого слова AS после вызова функции count:

```
CREATE OR REPLACE VIEW seats_by_fare_cond AS
SELECT a.model, s.aircraft_code, s.fare_conditions,
       count( * ) AS num_seats
FROM seats
GROUP BY aircraft_code, fare_conditions
ORDER BY aircraft_code, fare_conditions;
ОШИБКА: изменить имя столбца "count" на "num_seats" в
представлении  нельзя
```

Предложение OR REPLACE (2)

PostgresPro

- А дело в том, что при первоначальном создании этого представления третий столбец уже получил имя count (такое имя ему дала СУБД).
- Сначала следует удалить это представление, а затем создать его заново.

```
DROP VIEW seats_by_fare_cond;
CREATE OR REPLACE VIEW seats_by_fare_cond AS
  SELECT a.model, s.aircraft_code, s.fare_conditions,
         count( * ) AS num_seats
  ...
```

Задание списка имен столбцов представления

PostgresPro

Второй способ задания имен столбцов в представлении — с помощью списка их имен, заключенного в скобки:

```
DROP VIEW seats_by_fare_cond;
CREATE OR REPLACE VIEW seats_by_fare_cond
  ( code, fare_cond, num_seats )
  AS
  SELECT aircraft_code, fare_conditions, count( * )
  FROM seats
  GROUP BY aircraft_code, fare_conditions
  ORDER BY aircraft_code, fare_conditions;
```

Представление «Рейсы» (1)

PostgresPro

В базе данных «Авиаперевозки» создано представление «Рейсы» (flights_v), сконструированное на основе таблицы «Рейсы» (flights), но содержащее дополнительную информацию, а именно:

– подробные сведения об аэропорте вылета (departure_airport, departure_airport_name, departure_city);

– подробные сведения об аэропорте прибытия (arrival_airport, arrival_airport_name, arrival_city);

– местное время вылета, как плановое, так и фактическое (scheduled_departure_local, actual_departure_local);

– местное время прибытия, как плановое, так и фактическое (scheduled_arrival_local, actual_arrival_local);

–продолжительность полета, как плановая, так и фактическая (scheduled_duration, actual_duration).

Представление «Рейсы» (2)

PostgresPro

Описание атрибута	Имя атрибута	Тип PostgreSQL
Идентификатор рейса	flight_id	integer
Номер рейса	flight_no	char(6)
Время вылета по расписанию	scheduled_departure	timestamptz
Время вылета по расписанию, местное время в пункте отправления	scheduled_departure_local	timestamp
Время прилета по расписанию	scheduled_arrival	timestamptz
Время прилета по расписанию, местное время в пункте прибытия	scheduled_arrival_local	timestamp
Планируемая продолжительность полета	scheduled_duration	interval

Представление «Рейсы» (3)

Описание атрибута	Имя атрибута	Тип PostgreSQL
Код аэропорта отправления	departure_airport	char(3)
Название аэропорта отправления	departure_airport_name	text
Город отправления	departure_city	text
Код аэропорта прибытия	arrival_airport	char(3)
Название аэропорта прибытия	arrival_airport_name	text
Город прибытия	arrival_city	text
Статус рейса	status	varchar(20)
Код самолета, IATA	aircraft_code	char(3)

Представление «Рейсы» (4)

PostgresPro

Описание атрибута	Имя атрибута	Тип PostgreSQL
Фактическое время вылета	actual_departure	timestamptz
Фактическое время вылета, местное время в пункте отправления	actual_departure_local	timestamp
Фактическое время прилета	actual_arrival	timestamptz
Фактическое время прилета, местное время в пункте прибытия	actual_arrival_local	timestamp
Фактическая продолжительность полета	actual_duration	interval

Представление «Рейсы» (5)

PostgresPro

- В пассажирских авиаперевозках время в билетах указывается *местное*. Это касается и времени вылета, и времени прилета.
- Если пункты отправления и назначения находятся в различных часовых поясах, то время вылета будет привязано к одному часовому поясу, а время прилета — к другому.
- Поэтому в нашем представлении «Рейсы» (flights_v) предусмотрены четыре столбца, отображающие местное время: два из них относятся к пункту отправления — scheduled_departure_local и actual_departure_local, а два других относятся к пункту прибытия — scheduled_arrival_local и actual_arrival_local.
- В качестве типа данных для этих четырех столбцов выбран тип timestamp without time zone (сокращенно — просто timestamp), а не timestamp with time zone (timestamptz).
- Причина в том, что при выборе timestamptz время автоматически преобразовывалось бы при выводе данных к текущему часовому поясу, установленному на компьютере пользователя, а нам нужно сохранить его значения такими, какими они являются в пункте отправления и пункте назначения

Запрос к представлению «Рейсы» (1)

PostgresPro

```
\d flights_v
```

Утилита psql предлагает альтернативный — расширенный — способ вывода информации, который включается с помощью команды

```
\x
SELECT * FROM flights_v;
-[ RECORD 1 ]-------------+-------------------------
flight_id                 | 1
flight_no                 | PG0405
scheduled_departure       | 2016-09-13 13:35:00+08
scheduled_departure_local | 2016-09-13 08:35:00
scheduled_arrival         | 2016-09-13 14:30:00+08
scheduled_arrival_local   | 2016-09-13 09:30:00
scheduled_duration        | 00:55:00
departure_airport         | DME
departure_airport_name    | Домодедово
departure_city            | Москва
...
```

Запрос к представлению «Рейсы» (2)

PostgresPro

```
-[ RECORD 1 ]-------------+---------------------------
...
arrival_airport           | LED
arrival_airport_name      | Пулково
arrival_city              | Санкт-Петербург
status                    | Arrived
aircraft_code             | 321
actual_departure          | 2016-09-13 13:44:00+08
actual_departure_local    | 2016-09-13 08:44:00
actual_arrival            | 2016-09-13 14:39:00+08
actual_arrival_local      | 2016-09-13 09:39:00
actual_duration           | 00:55:00...
...
```

Удаление представления

PostgresPro

Бывают ситуации, когда заранее известно, что возможна попытка удаления несуществующего представления. В таких случаях обычно стараются избежать ненужных сообщений об ошибке отсутствия представления. Делается это путем добавления в команду DROP VIEW фразы IF EXISTS. Например:

```
DROP VIEW IF EXISTS flights_v;
```

PostgreSQL предлагает свое расширение — так называемое **материализованное представление**. Упрощенный синтаксис команды для создания материализованных представлений, таков:

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] table_name
[ (column_name [, ...] ) ]
AS query
[ WITH [ NO ] DATA ];
```

В квадратных скобках необязательные элементы команды

- Материализованное представление заполняется данными *в момент выполнения команды для его создания*, если только в команде не было фразы WITH NO DATA.
- Если же она была включена в команду, тогда в момент своего создания представление данными не заполняется, а для заполнения его данными нужно использовать команду

REFRESH MATERIALIZED VIEW

Материализованное представление очень похоже на обычную таблицу. Однако оно отличается от таблицы тем, что не только сохраняет данные, но также *запоминает запрос*, с помощью которого эти данные были собраны.

Материализованное представление «Маршруты» (1)

PostgresPro

Таблица «Рейсы» (flights) содержит избыточность: для одного и того же номера рейса, отправляющегося в различные дни, повторяются коды аэропортов отправления и назначения, а также код самолета. Таким образом, из этой таблицы можно извлечь информацию о маршруте, т. е. номер рейса, аэропорты отправления и назначения. Эта информация не зависит от конкретной даты вылета.

Описание атрибута	Имя атрибута	Тип PostgreSQL
Номер рейса	flight_no	char(6)
Код аэропорта отправления	departure_airport	char(3)
Название аэропорта отправления	departure_airport_name	text
Город отправления	departure_city	text
Код аэропорта прибытия	arrival_airport	char(3)

Материализованное представление «Маршруты» (2)

Описание атрибута	Имя атрибута	Тип PostgreSQL
Название аэропорта прибытия	arrival_airport_name	text
Город прибытия	arrival_city	text
Код самолета, IATA	aircraft_code	char(3)
Продолжительность полета	duration	interval
Дни недели, когда выполняются рейсы	days_of_week	integer[]

Если впоследствии вам потребуется обновить данные в материализованном представлении, то выполните команду

```
REFRESH MATERIALIZED VIEW routes;
```

Конечно, как и любой другой объект базы данных, материализованное представление можно удалить.

```
DROP MATERIALIZED VIEW routes;
```

Что дают представления? (1)

1. Упрощение разграничения полномочий пользователей на доступ к хранимым данным.
   Разным типам пользователей могут требоваться различные данные, хранящиеся в одних и тех же таблицах. Это касается как столбцов, так и строк таблиц. Создание различных представлений для разных пользователей избавляет от необходимости создавать дополнительные таблицы, дублируя данные, и упрощает организацию системы управления доступом к данным.
2. Упрощение запросов к базе данных.
   Использование представлений позволяет скрыть сложные запросы от прикладного программиста и сделать запросы более простыми и наглядными.

Что дают представления? (2)

3. Снижение зависимости прикладных программ от изменений структуры таблиц базы данных.
   Столбцы представления, т. е. их имена, типы данных и порядок следования, — это, образно говоря, интерфейс к запросу, который реализуется данным представлением. Если этот интерфейс остается неизменным, то SQL-запросы, в которых используется данное представление, корректировать не потребуется. Нужно будет лишь в ответ на изменение структуры базовых таблиц, на основе которых представление сконструировано, соответствующим образом перестроить запрос, выполняемый данным представлением.
4. Снижение времени выполнения сложных запросов за счет использования материализованных представлений.
   Если, например, какой-нибудь сводный отчет формируется длительное время, а запросы к отчету будут неоднократными, то может оказаться целесообразным сформировать его заранее и сохранить в материализованном представлении
   Недостаток материализованных представлений: необходимо своевременно обновлять с помощью команды REFRESH, чтобы они содержали актуальные данные.

3.5. Схемы базы данных

Общие сведения

PostgresPro

- **Схема** — это логический фрагмент базы данных, *в котором могут содержаться* различные объекты: таблицы, представления, индексы и др.
- В базе данных обязательно есть *хотя бы одна* схема.
- При создании базы данных в ней автоматически создается схема с именем **public**. Когда мы создавали таблицы в базе данных edu, они создавались именно в этой схеме.
- В каждой базе данных может содержаться более одной схемы. Их имена должны быть уникальными в пределах конкретной базы данных.
- Имена объектов базы данных (таблиц, представлений, последовательностей и др.) должны быть уникальными в пределах конкретной схемы, но *в разных схемах имена объектов могут повторяться*.
- Таким образом, можно сказать, что схема образует так называемое *пространство имен*.

Какие схемы у нас есть?

PostgresPro

Посмотреть список схем в базе данных можно так:

\dn

```
Список схем
    Имя    | Владелец
----------+----------
 bookings | postgres
 public   | postgres
(2 строки)
```

В учебной базе данных demo есть схема bookings. Все таблицы созданы именно в этой схеме. Для организации доступа к ней вы уже выполняли команду

SET search_path = bookings;

Если в базе данных несколько схем

PostgresPro

- Если в базе данных создано более одной схемы, то доступ к объектам, содержащимся в конкретной схеме, можно организовать разными способами. Первый заключается в том, чтобы имена объектов предварять именем схемы.

```
SELECT * FROM bookings.aircrafts;
```

- Другой способ заключается в том, чтобы одну из схем сделать текущей. В конфигурации сервера PostgreSQL есть параметр **search_path**. Его значение по умолчанию можно изменить в конфигурационном файле postgresql.conf. Он содержит имена схем, которые PostgreSQL просматривает при поиске конкретного объекта базы данных, когда имя схемы в команде не указано.

```
SHOW search_path;
   search_path
-----------------
 "$user", public
(1 строка)
```

Использование схем (1)

PostgresPro

- При наличии в параметре схемы **"$user"** *могут упроститься* некоторые операции с базой данных, если будут созданы схемы с именами, совпадающими с именами пользователей.
- Однако в базе данных demo нет таких схем, поэтому в результате все обращения к объектам базы данных без указания имени схемы будут адресоваться схеме public.
- Для изменения *порядка просмотра схем* при поиске объектов в базе данных служит команда SET. При этом первой в списке схем следует указать именно ту, которую СУБД должна просматривать первой. Эта схема и станет *текущей*.

```
SET search_path = bookings;
SHOW search_path;
 search_path
-------------
 bookings
(1 строка)
```

Использование схем (2)

PostgresPro

- Список может содержать боле одной схемы. Первая из них будет текущей.

```
SET search_path = bookings, public;
SELECT current_schema;
 current_schema
----------------
 bookings
(1 строка)
```

- По умолчанию объекты базы данных будут создаваться в текущей схеме. Если же нужно создать объект в конкретной схеме, которая не является текущей:

```
CREATE TABLE my_schema.airports
...
```

Литература

1. Лузанов, П. В. Postgres. Первое знакомство [Текст] / П. В. Лузанов, Е. В. Рогов, И. В. Лёвшин. – 5-е изд., перераб. и доп. – М. : Постгрес Профессиональный, 2019. – 156 с. https://edu.postgrespro.ru/introbook_v5.pdf
2. Моргунов, Е. П. PostgreSQL. Основы языка SQL [Текст] : учеб. пособие / Е. П. Моргунов ; под ред. Е. В. Рогова, П. В. Лузанова. – СПб. : БХВ-Петербург, 2018. – 336 с. https://edu.postgrespro.ru/sql_primer.pdf
3. Новиков, Б. А. Основы технологий баз данных [Текст] : учеб. пособие / Б. А. Новиков, Е. А. Горшкова ; под ред. Е. В. Рогова. – М. : ДМК Пресс, 2019. – 240 с. https://edu.postgrespro.ru/dbtech_part1.pdf
4. PostgreSQL [Электронный ресурс] : официальный сайт / The PostgreSQL Global Development Group. – https://www.postgresql.org.
5. Postgres Professional [Электронный ресурс] : российский производитель СУБД Postgres Pro : официальный сайт / Postgres Professional. – https://postgrespro.ru.

Задание

PostgresPro

Для выполнения практических заданий необходимо использовать книгу:

Моргунов, Е. П. PostgreSQL. Основы языка SQL [Текст] : учеб. пособие / Е. П. Моргунов ; под ред. Е. В. Рогова, П. В. Лузанова. – СПб. : БХВ-Петербург, 2018. – 336 с.

https://postgrespro.ru/education/books/sqlprimer

1. Изучить материал главы 5. Запросы к базе данных выполнять с помощью утилиты psql, описанной в главе 2, параграф 2.2.