

Язык SQL

Лекция 9

Программирование на стороне сервера

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

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)
```

- Назначьте схему bookings в качестве текущей

```
demo=# set search_path = bookings;
```

- Для останова сервера баз данных PostgreSQL служит команда
`pg_ctl stop -D /usr/local/pgsql/data -l postgres.log`
- Если у вас база данных demo была модифицирована, то для ее восстановления выполните команду
`psql -f demo_small.sql -U postgres`

9.1. Общая информация

- При разработке приложений с базами данных часто бывает целесообразно переложить часть операций с данными с клиентского приложения на *серверную часть СУБД*.
- Это позволяет
 - упростить программный код приложения,
 - уменьшить объем данных, передаваемых по сети и
 - ускорить работу приложения.

- Для расширения возможностей СУБД разрабатываются так называемые хранимые процедуры, которые в рамках PostgreSQL называются функциями.
- В версии 11 появились и хранимые процедуры.
- Такие функции можно писать на языке SQL (и не только).
- В функцию можно включать произвольные SQL-команды.
- Возвращаемым значением будет первая строка результата выполнения последней SQL-команды.
- Тело SQL-функции должно представлять собой список SQL-операторов, разделённых точкой с запятой. Точка с запятой после последнего оператора может отсутствовать.
- Если только функция не объявлена как возвращающая void, последним оператором должен быть SELECT, либо INSERT, UPDATE или DELETE с предложением RETURNING.

- Если функция не должна возвращать никакого полезного значения, тогда возвращаемым типом будет `void`.
- Функция может получать аргументы. Эти аргументы могут иметь модификаторы `IN` и `OUT`.
- Первый из них используется по умолчанию и означает параметр, значение которого будет изменяться внутри функции, но вне функции его новое значение не будет доступно.
- Параметр с ключевым словом `OUT` используется для того, чтобы вернуть из функции более одного значения.
- Парные знаки `$$` ограничивают непосредственно тело (определение) функции.
- Завершает определение функции наименование языка, на котором она написана.

Задача: подсчитать число мест в салоне самолета, соответствующих указанному классу обслуживания.

```
SET search_path = bookings;

DROP FUNCTION count_seats( char(3), text );

CREATE FUNCTION count_seats( a_code char(3),
                             fare_cond text )
    RETURNS bigint AS $$
    SELECT count( * ) FROM seats s
    WHERE s.aircraft_code = a_code AND
          s.fare_conditions = fare_cond;
$$ LANGUAGE sql;
```

- Сохраните этот текст в файле count_seats.sql.

```
psql -d demo -f count_seats.sql -U postgres
SET
DROP FUNCTION
CREATE FUNCTION
```


\x

Расширенный вывод включён.

\df

Список функций

```
-[ RECORD 1 ]-----+-----  
Схема                | bookings  
Имя                   | count_seats  
Тип данных результата | bigint  
Типы данных аргументов | a_code character, fare_cond text  
Тип                   | обычная
```

`\sf count_seats`

```
CREATE OR REPLACE FUNCTION bookings.count_seats(a_code
character, fare_cond text)
  RETURNS bigint
  LANGUAGE sql
AS $function$
SELECT count( * ) FROM seats s
WHERE s.aircraft_code = a_code AND
      s.fare_conditions = fare_cond;
$function$
```

```
SELECT count_seats( '773', 'Business' );
```

```
count_seats
```

```
-----
```

```
30
```

```
(1 строка)
```

```
SELECT count_seats( '763', 'Economy' );
```

```
count_seats
```

```
-----
```

```
192
```

```
(1 строка)
```

```
SELECT count_seats( 'CN1', 'Business' );
```

```
count_seats
```

```
-----
```

```
0
```

```
(1 строка)
```

Можно и так



```
SELECT * FROM count_seats( '763', 'Economy' );
```

- PostgreSQL допускает перегрузку функций; то есть позволяет использовать *одно имя* для нескольких различных функций, если у них *различаются типы* входных аргументов.
- Две функции считаются совпадающими, если они имеют одинаковые имена и типы *входных* аргументов, параметры OUT игнорируются.

Задача: вывести число мест для каждого класса обслуживания в салоне выбранной модели самолета.

```
SET search_path = bookings;
```

Такое же имя, а параметры другие



```
CREATE OR REPLACE FUNCTION count_seats( a_code char(3)
    DEFAULT 'SU9', OUT a_model text,
    OUT seats_business bigint,
    OUT seats_comfort bigint,
    OUT seats_economy bigint )
AS
$$
SELECT a.model,
( SELECT count( * )
  FROM seats s
  WHERE s.aircraft_code = a_code
        AND s.fare_conditions = 'Business'
) AS business,
...
```

```
...
( SELECT count( * )
  FROM seats s
  WHERE s.aircraft_code = a_code
        AND s.fare_conditions = 'Comfort'
) AS comfort,
( SELECT count( * )
  FROM seats s
  WHERE s.aircraft_code = a_code
        AND s.fare_conditions = 'Economy'
) AS economy
FROM aircrafts a
WHERE a.aircraft_code = a_code
ORDER BY 1;
$$ LANGUAGE sql;
```

```
psql -d demo -f count_seats2.sql -U postgres
```

```
SELECT * FROM count_seats();
```

без параметра
(по умолчанию)

```
-[ RECORD 1 ]--+------  
a_model      | Sukhoi SuperJet-100  
seats_business | 12  
seats_comfort  | 0  
seats_economy  | 85
```

```
SELECT * from count_seats( '319' );
```

с параметром

```
-[ RECORD 1 ]--+------  
a_model      | Airbus A319-100  
seats_business | 20  
seats_comfort  | 0  
seats_economy  | 96
```

```
SELECT * FROM count_seats( ( SELECT aircraft_code
                             FROM aircrafts
                             WHERE model =
                             'Boeing 777-300' ) );
```

получим значение параметра
с помощью подзапроса

```
-[ RECORD 1 ]--+-+-----
a_model       | Boeing 777-300
seats_business | 30
seats_comfort  | 48
seats_economy  | 324
```

двойные скобки!

```
SELECT seats_business + seats_comfort +
       seats_economy AS total_seats
FROM ( SELECT * from count_seats( '319' ) )
      AS groups_seats;
```

подзапрос в предложении FROM

```
total_seats
-----
          116
(1 строка)
```


- Заметьте, что выходные параметры не включаются в список аргументов при вызове такой функции из SQL.
- Это объясняется тем, что PostgreSQL определяет сигнатуру вызова функции, рассматривая только входные параметры.
- Это также значит, что при таких операциях, как удаление функции, в ссылках на функцию учитываются только типы входных параметров.
- Для удаления функции используется команда DROP FUNCTION. Например:
`DROP FUNCTION sum_and_product(int, int);`

Переменное число параметров (VARIADIC) (1)

Задача: расширить предыдущую задачу на переменное число моделей самолетов.

```
SET search_path = bookings;

CREATE OR REPLACE FUNCTION count_seats_var(
    VARIADIC a_codes char[] )
    RETURNS TABLE( model text, business bigint, comfort bigint,
                    economy bigint )
    AS
    $$
SELECT a.model,
( SELECT count( * )
  FROM seats s
  WHERE s.aircraft_code = a.aircraft_code
        AND s.fare_conditions = 'Business'
) AS business,
...
```

Переменное число параметров (VARIADIC) (2)

```
...  
( SELECT count( * )  
  FROM seats s  
  WHERE s.aircraft_code = a.aircraft_code  
        AND s.fare_conditions = 'Comfort'  
) AS comfort,  
( SELECT count( * )  
  FROM seats s  
  WHERE s.aircraft_code = a.aircraft_code  
        AND s.fare_conditions = 'Economy'  
) AS economy  
FROM aircrafts a  
WHERE a.aircraft_code IN ( SELECT unnest( a_codes ) )  
ORDER BY 1;  
$$ LANGUAGE sql;
```

←
разворачивает массив в виде
столбца таблицы

```
psql -d demo -f count_seats3.sql -U postgres
```

```
SELECT * FROM count_seats_var( '773', '319', 'CN1',  
                               'SU9' );
```

model	business	comfort	economy
Airbus A319-100	20	0	96
Boeing 777-300	30	48	324
Cessna 208 Caravan	0	0	12
Sukhoi SuperJet-100	12	0	85

(4 строки)

- Для удаления функции используется команда DROP FUNCTION.
Например:

```
DROP FUNCTION count_seats_var(VARIADIC a_codes char[]);
```

```
DROP FUNCTION
```

Как узнать, какие функции есть в базе данных?

`\dfn`

Список функций

```
-[ RECORD 1 ]-----+-----  
Схема          | bookings  
Имя            | count_seats  
Тип данных результата | record  
Типы данных аргументов | a_code character DEFAULT 'SU9'::bpchar,  
OUT a_model text, OUT seats_business bigint, OUT seats_comfort  
bigint, OUT seats_economy bigint  
Тип           | обычная  
-----+-----  
-[ RECORD 2 ]-----+-----  
Схема          | bookings  
Имя            | count_seats  
Тип данных результата | bigint  
Типы данных аргументов | a_code character, fare_cond text  
Тип           | обычная  
-----+-----  
-[ RECORD 3 ]-----+-----  
Схема          | bookings  
Имя            | count_seats_var  
Тип данных результата | TABLE(model text, business bigint,  
comfort bigint, economy bigint)  
Типы данных аргументов | VARIADIC a_codes character[]  
Тип           | обычная
```

- Для каждой функции определяется характеристика *изменчивости*, с возможными вариантами: VOLATILE, STABLE и IMMUTABLE. Если эта характеристика не задаётся явно в команде CREATE FUNCTION, по умолчанию подразумевается VOLATILE.
- Категория изменчивости представляет собой обещание некоторого поведения функции для оптимизатора.
- PostgreSQL требует, чтобы функции STABLE и IMMUTABLE не содержали SQL-команд, кроме SELECT, для предотвращения модификации данных.

- Изменчивая функция (VOLATILE) может делать всё, что угодно, в том числе, модифицировать базу данных. Она может возвращать *различные результаты* при нескольких вызовах с *одинаковыми аргументами*.
- Оптимизатор не делает никаких предположений о поведении таких функций. В запросе, использующем изменчивую функцию, она будет *вычисляться заново для каждой строки*, когда потребуются её результат.
- Характеристика VOLATILE (изменчивая) показывает, что результат функции может меняться даже в рамках одного сканирования таблицы, так что её вызовы нельзя оптимизировать. Изменчивы в этом смысле относительно немногие функции баз данных, например: `random()`, `currval()` и `timeofday()`.

- Стабильная функция (STABLE) *не может модифицировать* базу данных и гарантированно возвращает одинаковый результат, получая одинаковые аргументы, *для всех строк* в одном операторе.
- Эта характеристика позволяет оптимизатору заменить множество вызовов этой функции одним.
- В частности, выражение, содержащее такую функцию, можно безопасно использовать в условии поиска по индексу.
- Так как при поиске по индексу целевое значение вычисляется только один раз, а не для каждой строки, использовать функцию с характеристикой VOLATILE в условии поиска по индексу нельзя.

- Постоянная функция (IMMUTABLE) не может модифицировать базу данных и гарантированно всегда возвращает *одинаковые результаты* для одних и тех же аргументов.
- Эта характеристика позволяет оптимизатору предварительно вычислить функцию, когда она вызывается в запросе с постоянными аргументами. Например, запрос вида `SELECT ... WHERE x = 2 + 2` можно упростить до `SELECT ... WHERE x = 4`, так как нижележащая функция оператора сложения помечена как IMMUTABLE.
- Если функция имеет такую характеристику, любой её вызов с аргументами-константами можно немедленно заменить значением функции

9.2. Триггеры

- **Триггер** – это механизм, заставляющий СУБД выполнить конкретную функцию, когда выполняется определенный тип операций.
- Триггеры могут быть связаны с таблицами и представлениями (views).
- Триггеры, связанные с таблицами, могут выполняться как ДО (BEFORE), так и ПОСЛЕ (AFTER) операций INSERT, UPDATE, DELETE.
- Если случается конкретное событие, приводящее к срабатыванию триггера, то вызывается так называемая триггерная функция, которая и обрабатывает это событие.
- Триггеры, связанные с представлениями (views), выполняются ВМЕСТО операций INSERT, UPDATE, DELETE.
- Для создания триггера сначала нужно создать триггерную функцию. Эта функция не должна принимать никаких аргументов и должна возвращать значение типа **trigger**. Необходимые ей данные триггерная функция получает от СУБД без участия программиста.

- Триггеры могут быть двух типов с точки зрения числа повторных вызовов триггерной функции:
 - функция может вызываться *для каждой строки*, на которую влияет команда, вызвавшая срабатывание триггера;
 - функция может вызываться *только один раз*, независимо от числа строк, подвергшихся воздействию команды, вызвавшей срабатывание триггера. Даже если таких строк не будет ни одной, триггерная функция вызывается все равно.
- Триггеры первого типа называют триггерами уровня строки (row-level), а триггеры второго типа – триггерами уровня команды (statement-level).
- Триггерные функции, вызываемые триггерами уровня SQL-команды, должны всегда возвращать значение NULL. Триггерные функции, вызываемые триггерами уровня строки (row-level), могут возвращать строку таблицы, если это необходимо с точки зрения логики этой функции.

- Триггер уровня строки, выполняемый ДО операции, может принимать одно из двух решений:
 - он может вернуть значение NULL, чтобы предотвратить операцию с текущей строкой таблицы, для которой и вызван этот триггер (например, вставку, обновление или удаление строки таблицы);
 - при выполнении операций вставки или обновления триггер уровня строки может модифицировать вставляемую или обновляемую строку таблицы, поскольку именно строка, возвращаемая триггером, и будет вставлена в таблицу или обновлена в ней.
- Поэтому триггер такого типа, если не планируется отмена операции или модифицирование строки таблицы, должен вернуть неизмененную строку таблицы. В случае операции вставки (INSERT) и обновления (UPDATE) это будет специальное значение NEW, а в случае операции удаления строки (DELETE) это будет специальное значение OLD.
- Для триггеров уровня строки, выполняемых после операции, значение, возвращаемое триггерной функцией, просто игнорируется, поэтому они могут возвращать значение NULL.

- Если для одного и того же события, имеющего место в отношении данной таблицы, определено несколько триггеров, то они срабатывают в алфавитном порядке их имен. Если предшествующий BEFORE-триггер модифицирует строку таблицы, то последующий BEFORE-триггер получает на вход уже модифицированную строку. Если предыдущий BEFORE-триггер возвращает значение NULL, то операция над данной строкой таблицы отменяется и последующие BEFORE-триггеры не срабатывают.
- В типичном случае BEFORE-триггеры уровня строки используются для проверки или модифицирования данных, которые будут вставлены в таблицу или обновлены в ней. Например, такой триггер может использоваться для вставки значения текущего времени в поле типа timestamp или для проверки согласованности значений двух или более полей текущей строки таблицы.
- AFTER-триггеры логично использовать для продвижения изменений, сделанных в текущей таблице, в другие таблицы или выполнять проверки согласованности данных с другими таблицами.

- Упрощенный синтаксис команды для создания триггера таков:

```
CREATE TRIGGER имя_триггера { BEFORE | AFTER | INSTEAD OF  
} { событие [ OR ... ] }
```

```
ON имя_таблицы
```

```
[ FOR [ EACH ] { ROW | STATEMENT } ]
```

```
EXECUTE PROCEDURE имя_функции ( аргументы )
```

- где «событие» – это одно из:

```
INSERT
```

```
UPDATE [ OF имя_столбца [, ... ] ]
```

```
DELETE
```


```
TRUNCATE
```

- Создадим две копии таблицы «Самолеты» (aircrafts).
- Первая будет предназначена для хранения данных, взятых из таблицы-прототипа, а вторая будет использоваться в качестве журнальной таблицы.

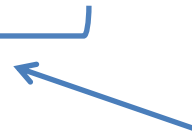
```
CREATE TABLE aircrafts_tmp AS  
  SELECT * FROM aircrafts WITH NO DATA;
```

```
ALTER TABLE aircrafts_tmp  
  ADD PRIMARY KEY ( aircraft_code );
```

```
ALTER TABLE aircrafts_tmp  
  ADD UNIQUE ( model );
```



Ограничения не создаются при копировании таблицы



данные не копируем


```
CREATE TABLE aircrafts_log AS
  SELECT * FROM aircrafts WITH NO DATA;
ALTER TABLE aircrafts_log
  ADD COLUMN when_add timestamp;
ALTER TABLE aircrafts_log
  ADD COLUMN operation text;
```

данные не копируем

когда выполнена операция

INSERT,
UPDATE или
DELETE

Задача: скопировать в таблицу `aircrafts_tmp` все данные из таблицы `aircrafts`, фиксируя все изменения в журнале изменений.

```
SET search_path = bookings;

CREATE OR REPLACE FUNCTION log_aircrafts()
RETURNS trigger AS
$$
BEGIN
    INSERT INTO aircrafts_log ( aircraft_code, model, range,
                               when_add, operation )
    VALUES ( NEW.aircraft_code, NEW.model, NEW.range,
             CURRENT_TIMESTAMP, 'INSERT' );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Язык PL/pgSQL



```
DROP TRIGGER IF EXISTS aircrafts_log ON aircrafts_tmp;
```

```
CREATE TRIGGER aircrafts_log AFTER INSERT ON aircrafts_tmp
FOR EACH ROW EXECUTE PROCEDURE log_aircrafts();
```

Сохраним в файле `log_aircrafts.sql`

```
psql -d demo -f log_aircrafts.sql -U postgres
```

```
SET
```

```
CREATE FUNCTION
```

```
DROP TRIGGER
```

```
CREATE TRIGGER
```

```
INSERT INTO aircrafts_tmp SELECT * FROM aircrafts;
```

```
INSERT 0 9
```

```
SELECT * FROM aircrafts_log;
```

aircraft_code	model	range	when_add	operation
773	Boeing 777-300	11100	2018-10-10	INSERT
763	Boeing 767-300	7900	2018-10-10	INSERT
...				

(9 строк)

9.3. Язык PL/pgSQL

- PL/pgSQL – это процедурный язык СУБД PostgreSQL. Он может использоваться для создания обычных функций и триггерных функций.
- Этот язык позволяет дополнить язык SQL управляющими структурами. С его помощью можно выполнять сложные вычисления.
- Функции, написанные на этом языке, могут использоваться везде, где могли бы использоваться встроенные функции языка SQL, например, в индексных выражениях при создании индексов.
- Данный язык позволяет повысить эффективность работы приложения с базой данных за счет того, что в рамках одной процедуры, написанной на этом языке, могут быть сгруппированы несколько SQL-операторов, которые хранятся на сервере.
- Поэтому клиентскому приложению не требуется выполнять эти SQL-операторы по одному, организуя каждый раз взаимодействие с сервером и тем самым увеличивая сетевой трафик.
- Также не выполняется передача промежуточных результатов вычислений от сервера к клиенту, тем самым также сокращается число взаимодействий клиента и сервера, что позволяет ускорить обработку данных.

- Функции на языке PL/pgSQL оформляются в виде блоков (в квадратных скобках указаны необязательные элементы):

```
[ <<метка>> ]
```

```
[ DECLARE
```

```
объявления ]
```

```
BEGIN
```

```
операторы
```

```
END [ метка ] ;
```

- Внутри блока могут содержаться вложенные блоки, которые удобно использовать для отражения логической структуры функции. Переменные, объявленные во вложенном блоке, скрывают одноименные переменные, объявленные во внешнем блоке.
- Все ключевые слова являются нечувствительными к регистру символов, поэтому их можно вводить как в верхнем, так и в нижнем регистре.

Пример функции, представленный в документации (1)

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
-- Объявим переменную типа integer и инициализируем ее.
quantity integer := 30;
BEGIN
-- Этот оператор выведет сообщение, в котором вместо знака %
-- будет подставлено значение переменной quantity, равное 30.
RAISE NOTICE 'Quantity here is %', quantity;
quantity := 50; -- присвоим переменной новое значение
--
-- Создадим вложенный блок.
--
DECLARE
-- Объявим переменную типа integer и инициализируем ее.
-- Имя этой переменной такое же, как и переменной в главном
-- блоке.
quantity integer := 80;
```

Пример функции, представленный в документации (2)

```
BEGIN
  -- Этот оператор выведет значение 80.
  RAISE NOTICE 'Quantity here is %', quantity;
  -- Этот оператор выведет значение 50. Поскольку имени
  -- переменной предшествует имя метки внешнего блока, будет
  -- использована переменная quantity из внешнего блока
  -- outerblock.
  RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;
END;
-- Вложенный блок завершился, значит, эта команда выведет
-- значение переменной, объявленной в главном блоке, т. е. 50.
RAISE NOTICE 'Quantity here is %', quantity;
RETURN quantity; -- возвратим результат
END;
$$ LANGUAGE plpgsql;
```


- В случае возникновения ошибки при выполнении функции PL/pgSQL работа функции прерывается. Но можно перехватывать возникающие ошибки и обрабатывать их тем или иным образом. Для этого в блок BEGIN...END вводится ключевое слово EXCEPTION.

```
[ <<label>> ]  
[ DECLARE  
  объявления ]  
BEGIN  
  операторы  
EXCEPTION  
  WHEN условие [ OR условие ... ] THEN  
    операторы_для_обработки_ошибки  
  [ WHEN условие [ OR условие ... ] THEN  
    операторы_для_обработки_ошибки  
  . ]  
END;
```

- Все условия, используемые в блоке обработки ошибок, имеют стандартизированные имена, приведенные в приложении А к документации на PostgreSQL.
- Если для возникшей ошибки предусмотрено соответствующее условие в данном блоке BEGIN..END, тогда эта ошибка обрабатывается здесь.
- Если же для нее обработчик не предусмотрен в данном блоке, тогда ошибка продвигается во внешний блок и обрабатывается там.
- Если же там обработчика для данной ошибки также нет, тогда выполнение функции прерывается.

- Для вывода сообщений пользователю или для генерирования ошибок служит команда RAISE. Покажем один из вариантов ее син-таксиса.

```
RAISE [ уровень ] 'формат' [, выражение [, ... ]];
```

- Здесь уровень означает степень серьезности сообщения: DEBUG, LOG, INFO, NOTICE, WARNING и EXCEPTION. По умолчанию используется EXCEPTION, что означает формирование ошибки. Параметр 'формат' служит для формирования текста сообщения, за этим параметром могут следовать переменные, значения которых подставляются в строку 'формат' в те позиции, которые обозначены символом «%». Приведем простой пример:

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

Задача: обеспечить равенство значения полной стоимости бронирования в таблице «Бронирования» (bookings) сумме стоимостей отдельных перелетов, которые были оформлены в рамках этой процедуры бронирования.

- Создадим новое бронирование и оформим один билет с двумя перелетами в нем.
- После завершения ввода строк в таблицу «Перелеты» мы проверим значение поля `total_amount`: оно станет равным сумме стоимостей всех забронированных перелетов.

```
SET search_path = bookings;

CREATE OR REPLACE FUNCTION update_bookings ()
RETURNS trigger AS
$$
DECLARE
    delta bookings.total_amount%TYPE;
    tick_no ticket_flights.ticket_no%TYPE;
BEGIN
    IF TG_OP = 'INSERT' THEN
        delta = NEW.amount;
        tick_no = NEW.ticket_no;

    ELSIF ( TG_OP = 'UPDATE' ) THEN
        delta = NEW.amount - OLD.amount;
        tick_no = OLD.ticket_no;

    ELSIF ( TG_OP = 'DELETE' ) THEN
        delta = OLD.amount * ( -1 );
        tick_no = OLD.ticket_no;
    END IF;
...

```

```
...  
    UPDATE bookings b SET total_amount = total_amount + delta  
    FROM tickets t, ticket_flights tf  
    WHERE b.book_ref = t.book_ref AND  
           t.ticket_no = tick_no;  
    RETURN NULL;  
END;  
$$ LANGUAGE plpgsql;  
  
DROP TRIGGER IF EXISTS update_bookings ON ticket_flights;  
  
CREATE TRIGGER update_bookings  
AFTER INSERT OR UPDATE OR DELETE ON ticket_flights  
FOR EACH ROW EXECUTE PROCEDURE update_bookings();
```

Проверяем в работе (подготовим транзакцию)

```
SET search_path = bookings;

BEGIN;

INSERT INTO bookings ( book_ref, book_date, total_amount )
VALUES ( 'ABC123', bookings.now(), 0 );

INSERT INTO tickets ( ticket_no, book_ref, passenger_id,
                    passenger_name)
VALUES ( '9991234567890', 'ABC123', '1234 123456',
        'IVAN PETROV' );

INSERT INTO ticket_flights ( ticket_no, flight_id,
                             fare_conditions, amount )
VALUES ( '9991234567890', 5572, 'Business', 12500 ),
        ( '9991234567890', 13881, 'Economy', 8500 );

COMMIT;

SELECT * from bookings WHERE book_ref = 'ABC123';
```

- Сохранить в файле check_update_bookings.sql

```
psql -d demo -f check_update_bookings.sql -U postgres
```

```
SET
```

```
BEGIN
```

```
INSERT 0 1
```

```
INSERT 0 1
```

```
INSERT 0 2
```

```
COMMIT
```

book_ref	book_date	total_amount
ABC123	2016-10-13 21:00:00+07	21000.00

(1 строка)



= 12 500 + 8 500


```
UPDATE ticket_flights SET amount = amount - 500
WHERE ( ticket_no, flight_id ) =
      ( '9991234567890', 5572 );
```

UPDATE 1

```
SELECT * from bookings WHERE book_ref = 'ABC123';
```

book_ref	book_date	total_amount
ABC123	2016-10-13 21:00:00+07	20500.00

(1 строка)

= 21 000 - 500



```
DELETE FROM ticket_flights
WHERE ( ticket_no, flight_id ) = ( '9991234567890', 5572 )
RETURNING *;
```

ticket_no	flight_id	fare_conditions	amount
9991234567890	5572	Business	12000.00

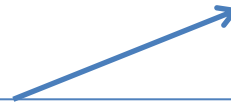
(1 строка)

```
DELETE 1
```

```
SELECT * from bookings WHERE book_ref = 'ABC123';
```

book_ref	book_date	total_amount
ABC123	2016-10-13 21:00:00+07	8500.00

(1 строка)



= 20 500 - 12 000

- Переменные в языке PL/pgSQL могут иметь любой тип данных, имеющийся в PostgreSQL, например integer, varchar и т. д.

DEFAULT означает присваивание :=

```
quantity integer DEFAULT 32;
```

- строковое значение нужно заключить в одинарные кавычки

```
url varchar := 'http://mysite.com';
```

- можно создать константу и инициализировать ее

```
user_id CONSTANT integer := 10;
```

- переменная для хранения значения поля user_id из таблицы users, Такой оператор избавляет нас от необходимости знать тип данных этого поля


```
user_id users.user_id%TYPE;
```

Задача: вывести сведения о числе проданных билетов и степени загрузки самолетов, выполняющих рейсы по указанному направлению: город отправления – город прибытия.

```
SET search_path = bookings;
```


```
CREATE OR REPLACE FUNCTION get_route_info( d_city text,
                                           a_city text )
  RETURNS TABLE( dep_city text, arr_city text,
                 flight_no char(6), flight_id integer,
                 scheduled_departure timestampz, model text,
                 total_seats integer, booked_seats integer,
                 percentage numeric )
```

возвращаем
таблицу



```
AS
$$
DECLARE
  tmp char(1);
  flight RECORD;
  tot_seats integer;
  b_seats integer;
  flights_found bool DEFAULT FALSE;
BEGIN
  ...
```

Тип RECORD. Ее
структура заранее не
определена



```
...
BEGIN
  IF NOT EXISTS ( SELECT 'x' FROM airports
                  WHERE city = d_city )
  THEN
    RAISE EXCEPTION 'Города % нет в базе данных', d_city;
  END IF;

  SELECT 'x' INTO tmp FROM airports WHERE city = a_city;
  IF NOT FOUND THEN
    RAISE NOTICE 'Города % нет в базе данных', a_city;
    RETURN;
  END IF;
```

проверка наличия города в базе данных,
выполненная по-разному в учебных целях

```
IF ( d_city = a_city ) THEN
  RAISE NOTICE 'Города отправления и прибытия ' ||
    'не должны совпадать';
  RETURN;
END IF;
```

...

-- Организуем цикл по результату запроса

```
FOR flight IN SELECT * FROM flights_v f, aircrafts a
      WHERE f.departure_city = d_city AND
      f.arrival_city = a_city AND
      f.aircraft_code = a.aircraft_code
```

LOOP

-- Для отладочных целей

-- RAISE NOTICE '% % % % %',


-- flight.departure_city, flight.arrival_city,

-- flight.flight_no, flight.flight_id,

-- flight.model;

-- Число мест в салоне самолета, выполняющего рейс

```
SELECT count(*) INTO tot_seats
FROM seats
WHERE aircraft_code = flight.aircraft_code;
```



переменная для
записи
результата

...


```
...
-- Число проданных билетов (перелетов)
  SELECT count(*) INTO b_seats
  FROM ticket_flights tf
  WHERE tf.flight_id = flight.flight_id;

  -- Формируется очередная строка результата
  RETURN QUERY SELECT flight.departure_city,
    flight.arrival_city,
    flight.flight_no, flight.flight_id,
    flight.scheduled_departure,
    flight.model,
    tot_seats, b_seats,
    round( ( b_seats::float /
            tot_seats::float )::
            numeric, 2 );

    flights_found = TRUE; -- строки были найдены
  END LOOP;
...

```

Выполнение функции
еще не завершается!



```
...  
-- Не было найдено ни одной строки  
IF NOT flights_found THEN  
    RAISE NOTICE 'Между городами % и % нет прямого рейса',  
                d_city, a_city;  
END IF;
```

Исключение,
сгенерированное в начале
функции, окажется здесь

Встроенная переменная,
содержащая текст
сообщения об ошибке

```
-- Обработка исключений  
EXCEPTION  
WHEN OTHERS THEN  
    RAISE NOTICE '%', SQLERRM;  
END;  
$$ LANGUAGE plpgsql;
```

Язык PL/pgSQL


```
SELECT * FROM get_route_info( 'Красноярск', 'Сочи' );
```

ЗАМЕЧАНИЕ: Города Красноярск нет в базе данных

```
SELECT * FROM get_route_info( 'Красноярск', 'Тверь' );
```

ЗАМЕЧАНИЕ: Города Тверь нет в базе данных

```
SELECT * FROM get_route_info( 'Москва', 'Москва' );
```

ЗАМЕЧАНИЕ: Города отправления и прибытия не должны совпадать

```
SELECT * FROM get_route_info( 'Красноярск', 'Анадырь' );
```

ЗАМЕЧАНИЕ: Между городами Красноярск и Анадырь нет прямого рейса

```
SELECT * FROM get_route_info( 'Сочи', 'Москва' );
```

dep_city		Сочи
arr_city		Москва
flight_no		PG0013
flight_id		30575
scheduled_departure		2016-10-26 21:15:00+07
model		Boeing 777-300
total_seats		402
booked_seats		316
percentage		0.79

-----+-----		
dep_city		Сочи
arr_city		Москва
flight_no		PG0013
flight_id		30576
scheduled_departure		2016-10-10 21:15:00+07
model		Boeing 777-300
total_seats		402
booked_seats		287
percentage		0.71

...

```
\df
```

```
...
```

```
-----  
Схема          | bookings  
Имя            | log_aircrafts  
Тип данных результата | trigger  
Типы данных аргументов |  
Тип           | триггерная  
-----+-----  
Схема          | bookings  
Имя            | update_bookings  
Тип данных результата | trigger  
Типы данных аргументов |  
Тип           | триггерная
```

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. Моргунов, Е. П. Администрирование информационных систем [Текст] : учеб. пособие / Е. П. Моргунов, О. Н. Моргунова ; Сиб. гос. аэрокосмич. ун-т. – Красноярск, 2015. – 90 с. <http://www.morgunov.org/programming.html>
4. 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
5. Учебный курс для разработчиков приложений PostgreSQL / Е. В. Рогов, П. В. Лузанов ; Postgres Professional. – <https://postgrespro.ru/education/courses>.
6. PostgreSQL [Электронный ресурс] : официальный сайт / The PostgreSQL Global Development Group. – <https://www.postgresql.org>.
7. Postgres Professional [Электронный ресурс] : российский производитель СУБД Postgres Pro : официальный сайт / Postgres Professional. – <https://postgrespro.ru>.

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. Подумать, в каких ситуациях, имеющих место при использовании базы данных «Авиаперевозки», можно было бы применить полученные знания.