Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский государственный технический университет имени   
Н.Э. Баумана

(национальный исследовательский университет)»

(МГТУ им. Н.Э. Баумана)



**УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ К ЛАБОРАТОРНЫМ РАБОТАМ**

**ПО КУРСУ «БАЗЫ ДАННЫХ»**

# Лабораторная работа №3 «Пользовательские функции и процедуры»

Авторы:

Кудрявцев А.П., [kudryavtsevap@bmstu.ru](mailto:kudryavtsevap@bmstu.ru)

Фомин М.М.

Москва, 2025

# Общие сведения

## Сокращения

SQL – Structured Query Language («язык структурированных запросов»).

БД – База данных.

СУБД – Система управления базами данных.

PL/pgSQL – Procedural Language/PostGres Structured Query Language.

Лабораторная работа посвящена изучению и разработке хранимых функций и процедур (одним словом, подпрограмм). Основная цель использования подпрограмм - упростить разрабатываемый проект за счет разбивки на более мелкие задачи. Упрощения достигаются за счет того, что эти задачи, став достаточно автономными, позволят определить интерфейс функции – параметры и возвращаемое значение. При этом, вызывающая сторона не видит тела функции и не зависит от ее изменения. Для разработки подпрограмм используется язык SQL и PL/pgSQL.

Лабораторная работа посвящена упрощению работы с помощью создания функций на языке SQL и PL/pgSQL.

PL/pgSQL – процедурное расширение языка SQL, используемое в СУБД PostgreSQL. Этот язык предназначен для написания функций, триггеров и правил и обладает следующими особенностями:

* Добавляет управляющие конструкции к стандарту SQL;
* Допускает сложные вычисления;
* Может использовать все объекты БД, определенные пользователем;
* Прост в использовании.

Данная лабораторная работа призвана сформировать у студента понимание создания алгоритмов для автоматизации процессов в БД и оптимизации разработки интерфейса.

# Лабораторная работа №3. Пользовательские функции и процедуры

Данная лабораторная состоит из двух частей. В каждой части есть теоретическая и практическая части лабораторной работы. В конце каждой части есть задания для выполнения в рамках лабораторной работы.

**Цель.**

Данная лабораторная работа призвана сформировать у студента понимание назначения пользовательских функции и процедур, их написание и использование.

**Задачи:**

* Получить теоретические знания о назначении функций и процедур БД.
* Изучить синтаксис функций и процедур.
* Научиться добавлять функции и процедуры в БД.
* Научиться удалять и изменять функции и процедуры.
* Научится использовать функции и процедуры.

# 1. Функции и процедуры на языке запросов (SQL)

## 1.1. Общие сведения

Основная цель использования функций - упростить разрабатываемый проект за счет разбивки на более мелкие задачи. Упрощения достигаются за счет того, что эти задачи, став достаточно автономными, позволят определить интерфейс функции – параметры и возвращаемое значение. При этом, вызывающая сторона не видит тела функции и не зависит от ее изменения.

Функции являются объектами БД, хранимыми в соответствующем системном каталоге, поэтому их называют хранимыми. Функция состоит из объявления – имени, необязательных аргументов, типа возвращаемого значения и тела функции. Тело функции хранится в виде строки и при вызове функции интерпретируется каждый раз.

Функция может использоваться только в контексте SQL – выражений, таких как SELECT, INSERT, UPDETE, FROM и т.п.

В функциях запрещено использовать команды управления транзакциями.

## 1.2. Функция с входными аргументами

Создадим функцию с одним аргументом, которая по номеру билета ticket\_no, являющимся первичным ключом, определяет пассажира. (Здесь и ниже используется демонстрационная БД «Авиаперевозки).

CREATE FUNCTION get\_passenger\_name( ticket\_no text) RETURNS text

AS $$

SELECT passenger\_name FROM tickets

WHERE ticket\_no = get\_passenger\_name.ticket\_no;

$$ LANGUAGE sql;

Вызвав функцию с выбранным значением, получим результат (рис. 1):

SELECT get\_passenger\_name( '0005432000991' );

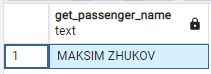


Рисунок 1 – Результат

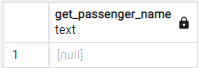
Рассмотрим функцию подробней. Имя функции может быть любое, но желательно, чтобы в имени был какой-то намек на то, что она делает. Ключевое слово RETURNS объявляет тип возвращаемого значения. Тело функции заключено в долларовые кавычки (тип кавычек может быть любым, но соответствовать правилам написания строковых констант). Функция может выполнять произвольное количество SQL операторов, но возвращает результат последнего запроса. Если оператор SELECT формирует несколько строк (и возвращаемое значение функции не SETOF), то будет возвращена первая строка. Поскольку поиск выполняется по первичному ключу, нам гарантирован возврат не более одного значения. Последним оператором в функции должен быть SELECT, либо INSERT, UPDATE или DELETE с предложением RETURNING.

Следует обратить внимание на то, что имя аргумента функции ticket\_no совпадает с именем колонки таблицы. Поэтому, чтобы их различить в предложении WHERE необходимо использовалась полное имя аргумента get\_passenger\_name.ticket\_no (все, как правило, стремятся сократить написание идентификаторов, но не следует это делать в ущерб читаемости программы, в данном случае можно сократить, например, задав имя аргумента «\_ticket\_no»).

Последняя строка «LANGUAGE sql» указывает, что использовался язык SQL запросов.

Если SELECT не найдет ни одной строки, то возвращается NULL.

SELECT get\_passenger\_name( '0' );



В описании функции не обязательно задавать имена аргументов, а использовать только тип аргументов и обращаться по номерам.

CREATE FUNCTION get\_passenger\_name(text) RETURNS text

AS $$

SELECT passenger\_name FROM tickets

WHERE ticket\_no = $1;

$$ LANGUAGE sql;

В примере вместо аргумента используется номер $1. Но так лучше не делать - ухудшает читаемость.

Создадим функцию, содержащую два аргумента.

CREATE FUNCTION is\_flights( IN airport\_name\_a text,   
 IN airport\_name\_b text DEFAULT 'Астрахань' ) RETURNS boolean

AS $$

SELECT EXISTS(SELECT \* FROM flights AS f

INNER JOIN airports AS aa ON aa.airport\_code =   
 f.arrival\_airport

INNER JOIN airports AS ad ON ad.airport\_code =   
 f.departure\_airport

WHERE aa.airport\_name = airport\_name\_a   
 AND ad.airport\_name = airport\_name\_b );

$$ LANGUAGE sql;

В объявлении функции мы использовали необязательное ключевое слово IN, обозначающее входной аргумент. Ключевое слово DEFAULT задает значение по умолчанию для аргумента. Функция проверяет есть ли рейсы между двумя аэропортами. Вызов функции с одним аргументом is\_flights('Шереметьево') или с двумя is\_flights ('Шереметьево', 'Астрахань') даст один и тот же результат.

Для всех функций рекомендуется задавать фактические параметры в том порядке, как они описаны в функции. Однако, при большо желании порядок аргументов можно изменить. Например, имеем функцию foo( arg1 integer, arg2 integer ), но используя следующую нотацию

foo(arg2 => <значение2>, arg1 => <значение1>) изменяем порядок фактических аргументов. Привычный вызов функции выглядел бы так - foo(<значение1>, <значение2>).

В заключении отметим, что функции могут не иметь аргументов, как например, приведенная ниже функция, определяющая количество аэропортов в БД.

CREATE FUNCTION get\_count\_airports() RETURNS integer

AS $$

SELECT count(\*) FROM airports;

$$ LANGUAGE sql;

## 1.3. Функции с выходными значениями

Как отмечалось выше, входные аргумента обозначаются ключевым словом IN. Кроме входных аргументов, функция может иметь выходные аргументы, обозначаемые, как OUT, а также входные-выходные - INOUT. Выходные аргументы определяют тип возвращаемых значений. Таким образом, можно определить возвращаемое значение двумя способами:

• с помощью ключевого слова RETURNS,

• через объявление выходных аргументов OUT или INOUT.

Можно использовать оба варианта одновременно, но тогда они должны быть согласованными, т.е. иметь один и тот же тип.

Перепишем функцию get\_count\_airports с использованием выходного параметра.

CREATE FUNCTION get\_count\_airports( OUT count integer )

AS $$

SELECT count(\*) FROM airports;

$$ LANGUAGE sql;

Вызов этой функции выполняется без указания выходного аргумента.

SELECT get\_count\_airports();

Дело в том, что выходные значения передаются как результат, а не посредством выходных аргументов (в отличие от многих языков программирования).

Рассмотрим функцию аналогичную get\_passenger\_name( ticket\_no text), но возвращающую два значения: имя пассажира и дату бронирования. Подчеркнем, что записать в RETURNS два возвращаемых значения нельзя.

CREATE FUNCTION get\_passenger\_name\_date( IN \_ticket\_no text, OUT name text, OUT date timestamp )

AS $$

SELECT t.passenger\_name, b.book\_date

FROM tickets AS t

LEFT OUTER JOIN bookings AS b ON b.book\_ref = t.book\_ref

WHERE t.ticket\_no = \_ticket\_no;

$$ LANGUAGE sql;

Дату бронирования мы получаем из таблицы bookings, используя внешний ключ book\_ref, запустив функцию.

SELECT get\_passenger\_name\_date( '0005432000991' );

Результат выполнения вполне ожидаемый. Получили пассажира и время выполнения бронирования (рис. 2):

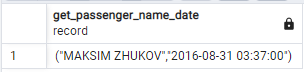


Рисунок 2 – Результат выполнения функции

Однако, если посмотреть внимательно, то увидим, во-первых, то, что тип возвращаемого значения даты timestamp, а тип поля book\_date в таблице bookings является дата с часовым поясом timestamptz. Нет строгого соответствия типа возвращаемого результата. Почему? Ответ на этот вопрос вы должны знать. Во-вторых, тип возвращаемого значения отмечен, как record. К этому вопросу мы вернемся после изучения составных типов.

В практической работе для обеспечения надежной работы функции следует проверять аргументы на возможность их использования. В частности, при выполнении расчетов не допускать деление на 0. Так же во многих случаях не понятно, что делать, если в функцию передан NULL. В этом случае надо завершить работу функции и в качестве результата вернуть NULL. Для этих целей в PostgreSQL имеется ключевое слово STRICT. Добавим STRICT в функцию get\_passenger\_name\_date( text ).

CREATE OR REPLACE FUNCTION get\_passenger\_name\_date(IN \_ticket\_no  
 text, OUT name text, OUT date timestamp)

AS $$

SELECT t.passenger\_name, b.book\_date

FROM tickets AS t

LEFT OUTER JOIN bookings AS b ON b.book\_ref=t.book\_ref

WHERE t.ticket\_no = \_ticket\_no;

$$ LANGUAGE sql STRICT;

Вызовем функцию с аргументом NULL.

get\_passenger\_name\_date( NULL );

На выходе получаем. При получении NULL функция сразу завешает работу и возвращает NULL (рис. 3):

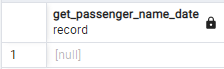


Рисунок 3 – Результат выполнения функции

В PostgreSQL функции являются объектами и идентифицируются они по имени и входным аргументам. Таким образом, функции foo( IN integer ) и foo( IN integer, OUT text ) не различаются и одновременно они не могут быть занесены в БД. Предположим, что хотим удалить хранимую функцию foo(IN integer, OUT text). Для этого мы должны записать функцию без выходных аргументов.

DROP FUNCTION foo (integer);

## 1.4. Категории изменчивости функций

Для каждой функции определена категория изменчивости, которая определяет изменяемость возвращаемого значения функции при одинаковых аргументах в запросах. Возможны следующие варианты изменчивости VOLATILE, STABLE и IMMUTABLE.

Функция VOLATILE (изменчивая) может делать что угодно, в том числе модифицировать БД. Может возвращать различные результаты при повторных вызовах с одинаковыми аргументами. Данная функция будет вычисляться каждый раз заново при обращении к ней, что исключает оптимизацию вызова этой функции. Если не указан тип изменчивости в функции, то предполагается по умолчанию VOLATILE.

Функция STABLE (стабильная) не модифицирует БД. Возвращаемое значение не изменяется в пределах одного SQL оператора при получении одинаковых значений аргументов. Это позволяет оптимизатору заменить несколько вызовов функции одним.

Функция IMMUTABLE (постоянная) не модифицирует БД. Возвращаемое значение будет всегда одним и тем же при одинаковых значениях аргументов. Это позволяет оптимизатору предварительно вычислить функцию в запросе.

Важно отметить контекст выполнения функций с разной категорией изменчивости. А именно, функция VOLATILE получает «свежий» снимок в начале каждого запроса, который он выполняет. Функции STABLE и IMMUTABLE используют снимок, полученный в начале вызывающего запроса. Поэтому, если функция содержит только команды SELECT, то ее следует указать как STABLE. Для функций, содержащих SELECT, не рекомендуется использовать IMMUTABLE, т.к. содержание таблиц может измениться. Однако, в некоторых случаях, когда таблицы редко меняются, будет целесообразным использовать IMMUTABLE. Например, функцию, рассчитывающая количество аэропортов (таблица airports), можно объявить, как IMMUTABLE. Вообще, IMMUTABLE можно ставить в функции, в которых возвращаемое значение зависит от аргументов.

По умолчанию функция get\_passenger\_name(…) создалась, как VOLATILE. Изменим категорию изменчивости на STABLE, используя оператор CREATE OR REPLACE

CREATE OR REPLACE FUNCTION get\_passenger\_name( ticket\_no text)  
 RETURNS text

AS $$

SELECT passenger\_name FROM tickets

WHERE ticket\_no = get\_passenger\_name.ticket\_no;

$$ STABLE LANGUAGE sql;

Категорию изменчивости функции можно изменить, используя оператор ALTER. Объявим функцию get\_passenger\_name\_date(text), как STABLE.

ALTER FUNCTION get\_passenger\_name\_date( IN \_ticket\_no text )   
 STABLE;

## 1.5. Составные типы

Составной тип представляет собой структуру, содержащую атрибуты (поля) и их тип. Составной тип можно рассматривать, как строку таблицы. Составной тип является объектом БД, регистрируем в системном каталоге. Определив тип, мы можем его использовать. PostgreSQL имеет достаточно своих составных типов и необходимость заведение новых составных типов возникает редко. Кроме того, при создании таблицы автоматически создается составной тип с тем же названием, представляющим строку этой таблицы. Составной тип можно использовать, как любой другой тип SQL.

БД «Авиаперелеты» имеется таблица Seats (места). В ней поле место seat\_no является символьным типом переменной длины varchar(4) и формат данных имеет вид <номер ряда><место>, например 10A, 10B, 10C и т.д.). В качестве примера мы заведем свой тип seat (место) и создадим новую таблицу с этим типом.

CREATE TYPE seat AS

(

line integer, -- ряд

place text -- место

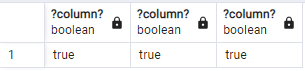
);

Перед созданием таблицы убедимся, что новый тип seat можно использовать в составном первичном ключе, в противном случае это может привести к большим накладным расходам и полному отказу от использования типа seat.

Должны работать операторы сравнения: <, >, = и т.д.

SELECT ROW(3, 'A')::seat = ROW(3, 'A')::seat,   
 '(1, "D")'::seat < '(1, "H")'::seat,   
 (1, 'B')::seat < (2, 'A')::seat;

Результат.



При проверке мы использовали три варианта задания составного типа. Ключевое слово ROW является конструктором табличной строки. Как видим, все работает.

Создадим таблицу.

CREATE TABLE seats\_my(

aircraft\_code char(3) NOT NULL,

seat\_no seat NOT NULL,

fare\_conditions varchar(10) NOT NULL,

CONSTRAINT seats\_my\_pkey PRIMARY KEY (aircraft\_code, seat\_no)

);

Как видите, первичный ключ содержит составной ключ. Вставим в таблицу несколько строк.

INSERT INTO seats\_my( aircraft\_code, seat\_no, fare\_conditions )

VALUES( '001', ROW( 10, 'A'), 'Economy' ),

( '001', ROW( 10, 'B'), 'Economy' ),

( '001', ROW( 10, 'C'), 'Economy' ),

( '001', ROW( 10, 'D'), 'Economy' ),

( '001', ROW( 10, 'E'), 'Economy' ),

( '001', ROW( 10, 'F'), 'Economy' ),

( '001', ROW( 11, 'A'), 'Economy' ),

( '001', ROW( 11, 'B'), 'Economy' ),

( '001', ROW( 11, 'C'), 'Economy' ),

( '001', ROW( 11, 'D'), 'Economy' ),

( '001', ROW( 11, 'E'), 'Economy' ),

( '001', ROW( 11, 'F'), 'Economy' );

Отметим, что можно не указывать явно поля таблицы в INSERT INTO seats\_my, а только указать имя таблицы, но это плохой стиль, особенно в хранимых функциях и процедурах. Дело в том, что БД, как правило дорабатываются, в таблицы вносят новые поля и, если явно не указывать список полей в INSERT, то некоторые значения будут занесены не в те поля таблицы. Посмотрим на результат (рис. 4):



Рисунок 4 – Результат вставки строк

Как видим, все прошло успешно, поле seat\_no представлено двумя атрибутами: номером и местом.

Теперь обратимся к атрибутам составного типа. Обращение seat\_no.line не будет работать, т.к. seat\_no воспринимается как имя таблицы. Надо имя составного типа взять в круглые скобки – (seat\_no).line. Напишем запрос.

SELECT aircraft\_code, (seats\_my.seat\_no).line, (seat\_no).place,  
 fare\_conditions

FROM seats\_my

WHERE (seat\_no).line > 10;

В запросе использовали два варианта получения доступа к атрибутам составного типа, в первом случае указали полное имя поля seats\_my.seat\_no, во втором – только составное поле. Как видим, атрибуты составного типа могут использоваться, как обычные скалярные поля таблицы, в WHERE, ORDER BY и т.д.

Результат запроса (рис. 5):

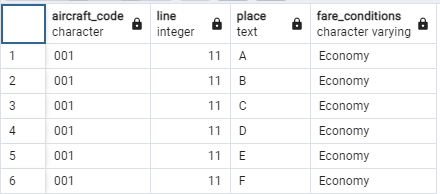
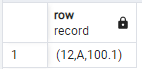


Рисунок 5 – Результат запроса

В PostgreSQL есть псевдотип record, являющимся неопределенным типом составного значения:

SELECT ( 12, 'A', 100.1 )::record;



Обратиться к атрибутам record невозможно, т.к. структура его не определена.

## 1.6. Функции и составные типы

Функция в качестве аргументов может принимать составной тип. Создадим функцию, которая возвращает место одной строкой.

CREATE FUNCTION get\_seat\_text( seat\_no seat ) RETURNS text

IMMUTABLE LANGUAGE sql

AS $$

SELECT seat\_no.line || seat\_no.place;

$$;

В этом примере изменен стиль написания функции, чтобы продемонстрировать такую возможность.

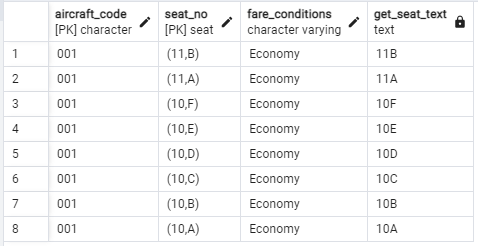
Вызовем функцию.

SELECT \*, get\_seat\_text( seat\_no )

FROM seats\_my

WHERE seat\_no < ( 11, 'C' )::seat

ORDER BY seat\_no DESC;



Напишем функцию, которая возвращает составной тип seat, получая отдельные атрибуты.

CREATE FUNCTION get\_seat(line integer, place text) RETURNS seat

IMMUTABLE LANGUAGE sql

AS $$

SELECT ROW( line, place )::seat;

$$;

или

CREATE FUNCTION get\_seat(line integer, place text) RETURNS seat

IMMUTABLE LANGUAGE sql

AS $$

SELECT line, place;

$$;

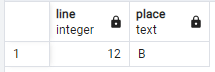
Проверим, дополнив пример суперпозицией функций.

SELECT get\_seat(12, 'B'), get\_seat\_text(get\_seat(12, 'B'));



Составной тип можно развернуть покомпонентно.

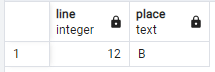
SELECT (get\_seat( 12, 'B' )).\*;



Имена столбцов соответствуют атрибутам составного типа seat.

Составной тип можно рассматривать, как однострочную таблицу, потому функцию можно использовать в предложении FROM.

SELECT \* FROM get\_seat( 12, 'B' );



Рассмотрим пример с использованием псевдотипа record.

CREATE FUNCTION get\_seat\_rec( line integer, place text ) RETURNS record

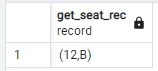
IMMUTABLE LANGUAGE sql

AS $$

SELECT line, place;

$$;

SELECT get\_seat\_rec( 12, 'B' );



Посмотрим, можно ли вызвать эту функцию и использовать покомпонентно в предложения SELECT и предложении FROM.

SELECT (get\_seat\_rec( 12, 'B' )).\*;

Сообщение.

ОШИБКА: тип записи не зарегистрирован SQL state: 42809

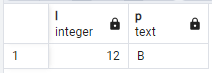
SELECT \* FROM get\_seat\_rec( 12, 'B' );

Сообщение.

ОШИБКА: у функций, возвращающих запись, должен быть список определений столбцов SQL state: 42601

Как видим, универсальность record имеет обратную сторону - ограниченная область применения, например, в представлении конечного результата. Ну а, если нам известна структура данных в record, то мы можем этим воспользоваться, явно задав ее в запросе. Типы значений атрибутов в record должны соответствовать задаваемой структуре rec(l integer, p text).

SELECT \* FROM get\_seat\_rec( 12, 'B' ) AS rec( l integer, p text );



## 1.7. Табличные функции

Табличные функции, это функции, возвращающие набор строк. Эти функции, естественно, вызывать в предложении FROM. Создадим функцию, которая возвращает места для заданного кода самолета (возвращаемся к использованию исходной таблицы seats).

CREATE FUNCTION get\_seats\_aircraft(aircraft\_code text)   
 RETURNS seats

STABLE LANGUAGE sql

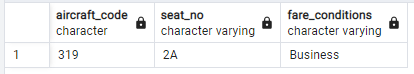
AS $$

SELECT \* FROM seats WHERE aircraft\_code = \_aircraft\_code;

$$;

Посмотрим, что получилось.

SELECT \* FROM get\_seats\_aircraft( '319');



А получили одну строку. И потому, что после RETURNS стоит не таблица seats, а составной тип seats, созданный PostgreSQL вместе с таблицей. В этом случае возвращается первая запись. Исправим функцию, добавив ключевое слово SETOF.

CREATE FUNCTION get\_seats\_aircraft( aircraft\_code text )   
 RETURNS SETOF seats

STABLE LANGUAGE sql

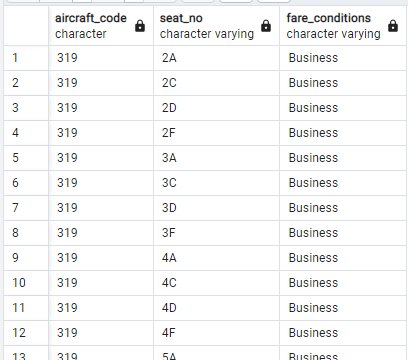
AS $$

SELECT \* FROM seats WHERE aircraft\_code = \_aircraft\_code;

$$;

В результате получаем все запрошенные записи.

SELECT \* FROM get\_seats\_aircraft( '319');



…

Вместо SETOF seats можно написать SETOF record. Как отмечалось выше, такая функция имеет ограниченное применение. Но, если будут использоваться выходные аргументы в функции в сочетании с record, то структура данных будет задана.

Например, подсчитаем количество мест в салоне самолета с учетом класса обслуживания.

CREATE FUNCTION get\_count\_seats\_aircraft(OUT aircraft\_code text,  
 OUT fare\_conditions text, OUT count integer)   
 RETURNS SETOF record

STABLE LANGUAGE sql

AS $$

SELECT aircraft\_code, fare\_conditions, count(\*)

FROM seats

GROUP BY aircraft\_code, fare\_conditions

ORDER BY aircraft\_code, fare\_conditions;

$$;

Выполним функцию. Как видим, мы можем с атрибутами записей, имена которых совпадают с выходными аргументами функции, выполнять различные операции.

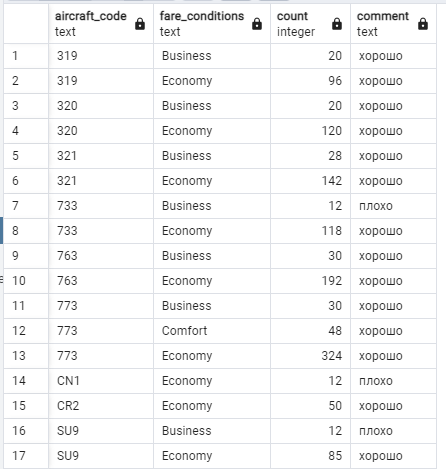
SELECT \*, CASE

WHEN count < 15 THEN 'плохо'

ELSE 'хорошо'

END AS comment

FROM get\_count\_seats\_aircraft();



Существует эквивалентная запись предыдущей функции с использованием ключевого слова TABLE.

CREATE FUNCTION get\_count\_seats\_aircraft()

RETURNS TABLE(aircraft\_code text, fare\_conditions text,   
 count integer)

STABLE LANGUAGE sql

AS $$

SELECT aircraft\_code, fare\_conditions, count( \* )

FROM seats

GROUP BY aircraft\_code, fare\_conditions

ORDER BY aircraft\_code, fare\_conditions;

$$;

# 1.8. Перегрузка функций

PostgreSQL позволяет определить несколько функций и процедур с одинаковым именем, но типы входящих (IN, INOUT) аргументов должны у функций отличаться. Такая возможность называется перегрузкой. При вызове функции PostgreSQL определяет по количеству и типу фактических аргументов какая функция подходит. Если подходящую функцию невозможно определить однозначно, то во время выполнения выдается ошибка.

Создадим функции, возвращающие наибольшее значения из заданных чисел.

CREATE FUNCTION maximum (a integer, b integer) RETURNS integer

IMMUTABLE LANGUAGE sql

AS $$

SELECT CASE WHEN a > b THEN a ELSE b END;

$$;

Запустим.

SELECT maximum( 20, 30 );



Запустим функцию с вещественными аргументами и получим ошибку.

SELECT maximum( 20.5, 30.7 );

ОШИБКА: функция maximum(numeric, numeric) не существует SQL state: 42883

Напишем еще одну функцию maximum, но с вещественными аргументами.

CREATE FUNCTION maximum(a double precision, b double precision)

RETURNS double precision

IMMUTABLE LANGUAGE sql

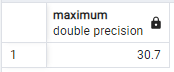
AS $$

SELECT CASE WHEN a > b THEN a ELSE b END;

$$;

Повторим попытку.

SELECT maximum( 20.5, 30.7 );



Определим наибольшее значение из трех целых чисел.

CREATE FUNCTION maximum (a integer, b integer, c integer) RETURNS integer

IMMUTABLE LANGUAGE sql

AS $$

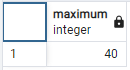
SELECT maximum (x, c) FROM maximum (a, b) max2(x);

$$;

В этом примере мы использовали функцию, определяющую максимум из двух целых чисел. А как знаем, вызов функции каждый раз интерпретируется, поэтому в данном случае, когда легко непосредственно определить наибольшее число, следует избегая вызовов функции.

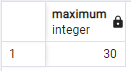
Проверим.

SELECT maximum( 40, 20, 30 );



Работает. Продолжим тестировать функцию (кстати говоря, при разработки любых программ тщательное тестирование избавит вас от многих проблем).

SELECT maximum( 40, NULL, 30 );



Неправильный результат! Дело в том, что NULL не является 0 для чисел или пробелом для символов, а является неопределенным значением. Любые операции с ним возвращают NULL. Когда передан в качестве фактического параметра NULL, то можно его интерпретировать, как 0, но это не следует делать, потому что, скорее всего используются не введенные данные. Исправим все функции maximum, добавив ключевое слово STRICT (о нем говорили выше), как в последнем примере.

CREATE FUNCTION maximum (a integer, b integer, c integer)  
 RETURNS integer

IMMUTABLE LANGUAGE sql STRICT

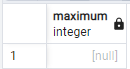
AS $$

SELECT maximum (x, c) FROM maximum (a, b) max\_ab(x);

$$;

Еще раз запустим.

SELECT maximum( 40, NULL, 30 );



Получение NULL в качестве результата является поводом для беспокойства. В нашем случае мы знаем, что некорректно переданы фактические аргументы.

К сожалению, функцию maximum надо написать и для других типов данных, меняя только объявление функции. Чтобы избежать этой непродуктивной работы, используем полиморфный тип.

## 1.9. Полиморфные функции

В качестве аргументов и возвращаемых значений для полиморфных функций используются полиморфные типы. Перечислим некоторые из них:

* anyelement – указывает, что функция принимает любой тип;
* anyarray – указывает, что функция принимает любой тип массива;
* anynoarray – указывает, что функция принимает любой тип, отличный от массива;
* anycompatible – указывает, что функция принимает аргументы, с автоматическим приведением несколько аргументов к общему типу.

Если функции возвращает полиморфный тип, то она должна иметь хотя бы один входящий аргумент полиморфного типа. Но полиморфная функция может возвращать фиксированный тип.

Перепишем функцию maximum с использованием полиморфного типа.

CREATE OR REPLACE FUNCTION maximum( a anyelement, b anyelement,  
 c anyelement ) RETURNS anyelement

IMMUTABLE LANGUAGE sql STRICT

AS $$

SELECT CASE

WHEN x > c THEN x

ELSE c END

END

FROM (SELECT CASE WHEN a > b THEN a ELSE b END) AS max\_ab(x);

$$;

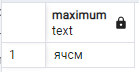
Проверим.

SELECT maximum( 40, 20, 30 ), maximum( 40.5, 20.3, 30.7 );



Посмотрим, будет ли функция работать со строками.

SELECT maximum( 'йцук'::text, 'ячсм', 'фыва' );



Работает. Теперь сравним целые и вещественные числа.

SELECT maximum( 40, 20.5, 30 );

ERROR: функция maximum(integer, numeric, integer) не существует

К сожалению, не работает. Причина в том, что фактические аргументы с типом anyelement должны иметь одинаковый тип. Как добиться естественного поведения функции? Для этого надо воспользоваться другим полиморфным типом – anycompatible. При использовании anycompatible выполняется попытка приведение типов к общему типу. Числовые типы приводятся к numeric без проблем. Перепишем функцию.

CREATE OR REPLACE FUNCTION maximum( a anycompatible,   
 b anycompatible, c anycompatible ) RETURNS anycompatible

IMMUTABLE LANGUAGE sql STRICT

AS $$

SELECT CASE

WHEN x > c THEN x

ELSE c END

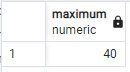
END

FROM (SELECT CASE WHEN a > b THEN a ELSE b END) AS max\_ab(x);

$$;

Повторим попытку.

SELECT maximum( 40, 20.5, 30 );



При задании числовых массивов в качестве фактических аргументов выполнение функции будет прервано сообщением об ошибке. Надо запретить использовать массивы, выбрав тип аргументов anycompatiblenoarray.

Как видим, и в перегрузке, и в полиморфизме есть свои сложности. Выбор способа при написании функций (процедур) решается в каждом конкретном случае.

## 1.10. Процедуры

Процедуры были введены в PostgreSQL 11. Основное отличие процедуры от функции состоит в том, что процедура может управлять транзакциями. Она может фиксировать или откатывать транзакцию во время выполнения. Процедуры, написанные на языке SQL, не могут использовать команды COMMIT и ROLLBACK. Такая возможность появляется у процедур, написанных на языке PL/pgSQL.

В отличии от функции, процедура не имеет предложение RETURNS. Но может иметь входные, выходные аргументы: IN, INOUT и OUT. Процедура вызывается командой CALL.

Создадим процедуру, добавляющую места для заданного тип самолета.

CREATE OR REPLACE PROCEDURE add\_seats\_my(\_aircraft\_code char(3),  
 line\_from integer,   
 line\_to integer,  
 \_fare\_conditions varchar(10) )

LANGUAGE sql

AS $$

INSERT INTO seats\_my( aircraft\_code, seat\_no, fare\_conditions )

SELECT \_aircraft\_code, ROW( line, chr( place+64 ) )::seat,

\_fare\_conditions

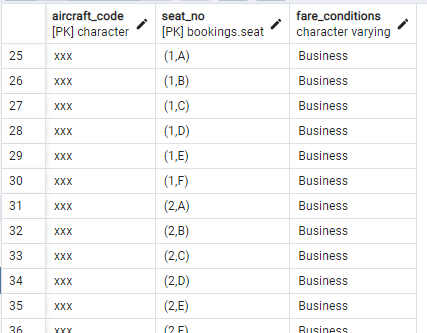
FROM generate\_series( line\_from, line\_to ) AS lines( line ),

generate\_series( 1, 6 ) AS places( place );

$$;

Вызовем процедуру.

CALL add\_seats\_my( 'xxx', 1, 10, 'Business');



…

# 2. Функции и процедуры на языке PL/pgSQL

## 2.1. Основы процедурного языка PL/pgSQL

PL/pgSQL - это процедурный язык для СУБД Postgres. Он используется для написания функций, процедур и триггеров. В отличие от языка запросов SQL, PL/pgSQL позволяет выполнять сложные вычисления, предоставляя возможность применения условных операторов, циклов, обработок ошибок, массивов и т.д.

PL/pgSQL это блочно-структурированный язык. Текст тела функции должен быть *блоком*. Структура блока:

[<<метка>>]

[DECLARE объявления]

BEGIN

*операторы*;

[EXCEPTION

*обработка ошибок*]

END [метка];

Объявления и операторы должны заканчиваться «;». В качестве операторов можно использовать команды PL/pgSQL и большинство команд SQL. Допускается вложенность блоков. Напомним, что квадратными скобками «[…]» отмечены необязательные элементы блока.

Структура функции выглядит следующим образом:

CREATE [OR REPLACE] FUNCTION имя\_функции(аргументы)

RETURNS тип\_возвращаемого\_значения

LANGUAGE plpgsql

AS $$

*блок*

$$;

Кроме использования функций и процедур, PL/pgSQL позволяет оформить операторы в анонимный блок с помощью команды DO.

DO $$

*блок*

$$;

В отличии от функций и процедур, анонимный блок не сохраняется на сервере, не имеет аргументов и не возвращает значения. Его целесообразно использовать для отладки функций и других вспомогательных целей.

Переменные PL/pgSQL могут иметь любой тип данных SQL, таких как integer, real, text, varchar. Кроме этого, PL/pgSQL предоставляет возможность расширить количество типов, ссылаясь на поля таблиц.

* Можно задать тип переменной, как у поля таблица, например:  
  **myfield aircrafts. aircraft\_code%TYPE;**  
  Такой способ задания типа переменной кажется излишним, на первый взгляд. Однако, имеет ряд преимуществ: не надо уточнять тип поля при его использовании (например, char (3) или varchar(10)), это раз. И второе, таблицы в БД не являются раз и на всегда заданными объектами, в них вносятся изменения, добавляются поля, изменяется размер полей и т.д. Предположим, что решили изменить тип поля aircraft\_code с char(3) на char(5). И если переменная myfield была описана, как char(3), то при присвоении myfield значения поля таблицы будут отброшены два символа и как результат, работающая функция перестает правильно работать.
* Можно задать тип переменной, как тип строки в таблице или курсора: **myrow aircrafts%ROWTYPE;**  
  Все сказанное выше, справедливо и здесь. Плюс то, что разумное применение этого объявления, делает код более компактным и читаемым.
* Можно задать тип переменной, как строка: **arow RECORD;**Переменные записи аналогичны переменным типа строки, но у них нет предопределенной структуры. Они принимают фактическую структуру строки, назначенной им во время команды SELECT или FOR. Структура переменной записи может меняться каждый раз, когда ей присваивается значение. Следствием этого является то, что до тех пор, пока переменная записи не будет впервые присвоена, она не имеет подструктуры, и любая попытка доступа к полю в ней приведет к ошибке времени выполнения.

Общий синтаксис объявления переменной:

*имя* [ CONSTANT ] *тип* [ COLLATE *имя\_правила\_сортировки* ] [ NOT NULL ] [ { DEFAULT | := | = } *выражение* ];

Предложение DEFAULT, если присутствует, задаёт начальное значение, которое присваивается переменной при входе в блок. Если отсутствует, то переменная инициализируется SQL-значением NULL. Указание CONSTANT предотвращает изменение значения переменной после инициализации, таким образом, значение остаётся постоянным в течение всего блока. Параметр COLLATE определяет правило сортировки, которое будет использоваться для этой переменной. Если указано NOT NULL, то попытка присвоить NULL во время выполнения приведёт к ошибке. Все переменные, объявленные как NOT NULL, должны иметь непустые значения по умолчанию. Можно

использовать знак равенства (=) вместо совместимого с PL/SQL :=.

Пример.

DO $$

DECLARE

val int NOT NULL = 0;

val2 CONSTANT int = 42;

BEGIN

val2 = val2+1; -- ошибка

END;

$$;

Рассмотрим вложенные блоки. Переменные внутри вложенного блока не видны внешнему блоку. А переменные внутреннего блока перекрывают переменные внешнего блока.

DO $$

<<m1>>

DECLARE

val int = 100;

BEGIN

<<m2>>

DECLARE

val int = 200;

BEGIN

RAISE NOTICE ' m1.val = %, m2.val = %', m1.val,  
 m2.val;

RAISE NOTICE 'val = %', val;

END m2;

END m1;

$$;

Результат.

ЗАМЕЧАНИЕ: m1.val = 100, m2.val = 200

ЗАМЕЧАНИЕ: val = 200

В примере используется команда RAISE NOTICE для вывода данных. Более подробно о команде RAISE смотрите в документации.

## 2.2. Функции PL/pgSQL

Многое сказанное о SQL функциях и процедур справедливо и для функций и процедур, написанных на PL/pgSQL:

* создание, изменение и удаление;
* аргументы;
* возвращаемые значения;
* категория изменчивости;
* перегрузка;
* полиморфизм;
* и т.д.

В отличие от SQL функций, функции PL/pgSQL возвращают значения посредством либо INOUT и OUT аргументов, либо оператором RETURN. Как видим, подобное поведение соответствует языкам программирования.

Рассмотрим примеры.

Результат возвращает оператор RETURN.

CREATE FUNCTION incr(IN a int) RETURNS int

LANGUAGE plpgsql IMMUTABLE

AS $$

DECLARE

inc\_val CONSTANT int = 10;

BEGIN

RETURN a+inc\_val;

END;

$$;

Результат возвращается аргументом OUT.

CREATE FUNCTION incr(IN a int, OUT ret int)

LANGUAGE plpgsql IMMUTABLE

AS $$

DECLARE

inc\_val CONSTANT int = 10;

BEGIN

ret = a+inc\_val;

END;

$$;

Результат возвращается аргументом INOUT.

CREATE FUNCTION incr(INOUT a int)

LANGUAGE plpgsql IMMUTABLE

AS $$

DECLARE

inc\_val CONSTANT int = 10;

BEGIN

a = a+inc\_val;

END;

$$;

## 2.3. Условные операторы

В PL/pgSQL есть два условных оператора: IF и CASE. Рассмотрим оператор IF.

IF *условие* THEN

*операторы*

ELSEIF *условие* THEN

*операторы*

…

ELSEIF *условие* THEN

*операторы*

ELSE

*операторы*

END IF;

Производится последовательная проверка условий и выполняются те операторы, которые соответствуют истинному условию. Слоев ELSEIF может быть несколько, либо они могут отсутствовать. Слой ELSE может отсутствовать. Напомню, что условие может принимать три значения: TRUE, FALSE и NULL. Например, оператор NULL<1 возвращает NULL.

Пример.

CREATE OR REPLACE FUNCTION comp\_ab( a int, b int ) RETURNS text

IMMUTABLE LANGUAGE plpgsql

AS $$

BEGIN

IF a IS NOT NULL AND b IS NOT NULL THEN

IF a < b THEN

RETURN format( '%s меньше %s', 'a', 'b' );

ELSEIF a > b THEN

RETURN format( '%s больше %s', 'a', 'b' );

ELSE

RETURN format( '%s равно %s', 'a', 'b' );

END IF;

ELSE

RETURN format('сравнение %s и %s не определено', 'a',  
 'b');

END IF;

END;

$$;

Проверим.

DO $$

DECLARE a int;

BEGIN

RAISE NOTICE '%, %, %', comp\_ab( 1, 2 ), comp\_ab( 3, 2 ),  
 comp\_ab( a, 2 );

END;

$$;

Результат.

ЗАМЕЧАНИЕ: a меньше b, a больше b, сравнение a и b не определено

Оператор CASE имеет две формы.

Первая форма.

CASE *выражение-поиска*

WHEN *выражение* [, *выражение* [...]] THEN

*операторы*

…

WHEN *выражение* [, *выражение* [...]] THEN

*операторы*

ELSE *операторы*

END CASE;

Слой WHEN может повторяться несколько раз. ELSE может отсутствовать. *Выражение-поиска* вычисляется один раз и последовательно сравнивается с каждым *выражением* в условиях WHEN. Если совпадение найдено, то выполняются соответствующие *операторы* и управление переходит к следующей после END CASE команде. Все последующие выражения WHEN не проверяются. Если совпадение не было найдено, то выполняются *операторы* в ELSE. Но если ELSE нет, то вызывается исключение CASE\_NOT\_FOUND.

Пример.

DO $$

DECLARE

val text = 'пять';

BEGIN

CASE val

WHEN 'один' THEN

RAISE NOTICE '%', 1;

WHEN 'два' THEN

RAISE NOTICE '%', 2;

WHEN 'три' THEN

RAISE NOTICE '%', 3;

WHEN 'четыре', 'пять' THEN

RAISE NOTICE '%', '4 или 5';

ELSE

RAISE NOTICE '%', 'больше пяти';

END CASE;

END;

$$;

Результат.

ЗАМЕЧАНИЕ: 4 или 5

Вторая форма.

CASE

WHEN *логическое-выражение* THEN

*операторы*

…

WHEN *логическое-выражение* THEN

*операторы*

ELSE *операторы*

END CASE;

Эта форма CASE реализует условное выполнение, основываясь на истинности логических условий. Каждое *логическое-выражение* в предложении WHEN вычисляется по порядку до тех пор, пока не будет найдено истинное. Затем выполняются соответствующие *операторы* и управление переходит к следующей после END CASE команде. (Все последующие выражения WHEN не проверяются.) Если ни одно из условий не окажется истинным, то выполняются *операторы* в ELSE. Но если ELSE нет, то вызывается исключение CASE\_NOT\_FOUND. Эта форма CASE полностью эквивалента IF-THEN-ELSIF, за исключением того, что при невыполнении всех условий и отсутствии ELSE, IF-THEN-ELSIF ничего не делает, а CASE вызывает ошибку.

Перепишем функцию comp\_ab с использованием CASE.

CREATE OR REPLACE FUNCTION comp\_ab( a int, b int ) RETURNS text

IMMUTABLE LANGUAGE plpgsql

AS $$

BEGIN

CASE

WHEN a IS NULL OR b IS NULL THEN

RETURN format( 'срвненине %s и %s не определено',  
 'a', 'b' );

WHEN a < b THEN

RETURN format( '%s меньше %s', 'a', 'b' );

WHEN a > b THEN

RETURN format( '%s больше %s', 'a', 'b' );

ELSE

RETURN format( '%s равно %s', 'a', 'b' );

END CASE;

END;

$$;

Проверим.

DO $$

DECLARE a int;

BEGIN

RAISE NOTICE '%, %, %', comp\_ab( 1, 2 ), comp\_ab( 3, 2 ),  
 comp\_ab( a, 2 );

END;

$$;

Результат.

ЗАМЕЧАНИЕ: a меньше b, a большеg b, срвненине a и b не определено

## 2.4. Циклы

В PL/pgSQL есть следящие варианты циклов: LOOP, WHILE, FOR FOREACH.

### 2.4.1. Цикл LOOP

Цикл LOOP имеет следующую конструкцию.

[<<*метка*>>]

LOOP

*операторы*

END LOOP [*метка* ];

Этот цикл выполняется бесконечно. Для выхода из цикла необходимо использовать операторы EXIT или RETURN. EXIT имеет следующий формат:

EXIT [*метка*] [WHEN *логическое-выражение*];

Циклы могут иметь вложенную структуру. С помощью указания метки в операторе EXIT можно завершить внешний цикл. Если метка не указана, то завершается текущий цикл. Если *логическое-выражение* в WHEN истинно цикл завершается. При отсутствии WHEN цикл завершается сразу при выполнении оператора EXIT.

Подсчитаем сумму чисел от 1 до 100.

DO $$

DECLARE

val int = 1;

sum int = 0;

BEGIN

LOOP

sum = sum+val;

val = val+1;

EXIT WHEN val = 101;

END LOOP;

RAISE NOTICE 'val=%, sum=%', val, sum;

END;

$$;

Результат.

ЗАМЕЧАНИЕ: val=101, sum=5050

### 2.4.2. **Цикл** WHILE

Цикл WHILE выполняется до тех пор, *логическое-выражение.*

[<<*метка*>>]

WHILE *логическое-выражение*

LOOP

*операторы*

END LOOP [*метка*];

Подсчитаем сумму чисел от 1 до 100 с использованием цикла WHILE.

DO $$

DECLARE

sum int = 0;

val int = 1;

BEGIN

WHILE val <= 100

LOOP

sum = sum+val;

val = val+1;

END LOOP;

RAISE NOTICE 'sum=%', sum;

END;

$$;

### 2.4.3. Цикл FOR

Рассмотрим целочисленный вариант цикла.

[<<*метка*>>]

FOR *имя* IN [REVERSE] *выражение-низ*..*выражение-верх* [BY *выражение-шаг*]

LOOP

*операторы*

END LOOP [*метка*];

Переменная *имя* (автоматически присваивается тип integer) последовательно принимает значения от *выражение-низ* до *выражение-верх* с шагом *выражение-шаг*. Если шаг не указан, то шаг итерации равен 1. Цикл завершается, когда переменная *имя* будет больше *выражение-верх*. Если указан REVERSE, то шаг не прибавляется, а вычитается, и цикл завершается, когда переменная *имя* будет меньше *выражение-верх*.

Область видимости переменной *имя* является цикл FOR, за пределами цикла переменная не видна.

Подсчитаем сумму чисел от 1 до 100 с использованием цикла FOR.

DO $$

DECLARE

sum int = 0;

BEGIN

FOR val IN 1..100

LOOP

sum = sum+val;

END LOOP;

RAISE NOTICE 'sum=%', sum;

END;

$$;

### 2.4.4. Операторы управления циклами EXIT и CONTINUE

Эти операторы применимы ко всем видам циклов. Оператор EXIT мы рассматривали выше, и он предназначен для выхода из цикла. Оператор CONTINUE - для завершения текущей итерации цикла и перехода к следующей итерации.

CONTINUE [*метка*] [WHEN *логическое-выражение*];

При наличии WHEN следующая итерация цикла начинается только тогда, когда *логическое-выражение* истинно. В противном случае управление переходит к оператору, следующему за CONTINUE. При отсутствии WHEN выполняется переход к следующей итерации.

Если *метка* не указана, то начинается следующая итерация текущего цикла. То есть все оставшиеся в цикле операторы пропускаются. Если *метка* присутствует, то она указывает на метку цикла, выполнение которого будет продолжено.

Подсчитаем сумму чисел от 1 до 100, без чисел кратных 10.

DO $$

DECLARE

sum int = 0;

BEGIN

FOR val IN 1..100

LOOP

CONTINUE WHEN mod(val, 10) = 0;

sum = sum+val;

END LOOP;

RAISE NOTICE 'sum=%', sum;

END;

$$;

## 2.5. Функции и SQL команды

Внутри функции и процедуры PL/pgSQL можно использовать SQL команды, не возвращающие результат, такие как INSERT, UPDATE, DELETE и т.д. Внутри SQL команд можно использовать переменные PL/pgSQL.

Команды COMMIT и ROLLBACK допускаются только в процедурах.

Пример использований SQL команд, не возвращающих результат.

CREATE FUNCTION fill\_mytab() RETURNS void

LANGUAGE plpgsql

AS $$

BEGIN

CREATE TABLE IF NOT EXISTS mytab

( id int,

data text

);

INSERT INTO mytab

VALUES (1, 'один'), (2, 'два'), (3, 'три');

END;

$$;

Выполним.

DO

$$

BEGIN

/\* ошибка

SELECT fill\_mytab();

\*/

PERFORM fill\_mytab();

END;

$$;

В PL/pgSQL, если SQL команда возвращает результат, то он должен быть куда-то помещен. В нашем случае непосредственный результат, возвращаемый запросом, нас не интересует (важен побочный результат, создание и заполнение таблицы). Для этих целей служит команда PERFORM.

## 2.6. Команды, возвращающие одну строку

Часто, требуется получить результат SQL команды в виде одной строки или набора скалярных значений. Если SQL запрос возвращает несколько строк, то используется только первая. Если не возвращается ни одна строка, то результат не определен. Случай, когда возвращается несколько строк или ни одной, говорит о том, что SQL команда написана неправильно. В PL/pgSQL предусмотрена проверка таких случаев. Результат запроса сохраняется с помощью предложения INTO.

SELECT *выражения\_select* INTO [STRICT] *цель* FROM ...;

INSERT ... RETURNING *выражения* INTO [STRICT] *цель*;

UPDATE ... RETURNING *выражения* INTO [STRICT] *цель*;

DELETE ... RETURNING *выражения* INTO [STRICT] *цель*;

*Цель* может быть переменной типа record, переменной составного типа или разделённым запятыми списком скалярных переменных. Количество и тип переменных в *выражения\_select*/*выражения* должны соответствовать переменным в *целях* (переменная record автоматически приводится в типу результата).

При указании STRICT (что рекомендуется) выполняется проверка на возвращение только одной строки. Если возвращается несколько строк, то формируется ошибка TOO\_MANY\_ROWS, если ни одной строки не возвращается - \_DATA\_FOUND.

В предыдущем разделе была с помощью функции fill\_mytab() создана таблица mytab и добавлены три записи. Запустим на выполнение эту функцию несколько раз, чтобы добиться дублирования строк таблицы. Рассмотрим пример извлечения данных одной строкой.

CREATE OR REPLACE FUNCTION get\_data( IN \_id int, OUT \_err text,  
 OUT \_data text )

STABLE LANGUAGE plpgsql

AS $$

BEGIN

SELECT data INTO STRICT \_data FROM mytab WHERE id = 1;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

\_err = SQLSTATE;

WHEN TOO\_MANY\_ROWS THEN

\_err = SQLSTATE;

END;

$$;

В функции get\_data имеется один входной аргумент \_id, по которому определяется нужная строка в таблице и два выходных текстовых параметра. Первый аргумент \_err предназначен для кода ошибки. Если исключение не сформируется, то значение \_err не определено (NULL). Второй аргумент будет содержать данные. В SELECT используется STRICT, поэтому, в таких случаях, может быть сформировано исключение, которое перехватывается и сохраняется код ошибка SQLSTATE. (Исключение можно обрабатывать в том модуле, в котором оно сформировано или в вызывающей подпрограмме.) Напишем процедуру do\_something, которая что-то делает и, в том числе, вызывает нашу функцию get\_data. В процедуре мы опять использовали конструкцию SELECT INTO для получения данных, возвращаемых функцией. Так как функция гарантировано возвращает одну строку, в STRICT нет надобности.

CREATE OR REPLACE PROCEDURE do\_something()

LANGUAGE plpgsql

AS $$

DECLARE

err text;

data text;

BEGIN

/\* Что-то делаем. \*/

SELECT \* INTO err, data FROM get\_data( 1 ) AS val(text,  
 text);

IF err IS NOT NULL THEN

RAISE NOTICE 'err=%', err;

ROLLBACK;

ELSE

COMMIT;

END IF;

END;

$$;

Получив данные, проверяем код ошибки. Если ошибка есть, то печатаем код ошибки и откатываем изменения.

Запустим процедуру.

CALL do\_something();

Как и следовало ожидать, вызов функции завершился, строка данных не получена.

ЗАМЕЧАНИЕ: err=P0003;

Обратим внимание на то, что возврат функцией ошибки не означает аварийное завершение. Выполнение процедуры продолжается и завершается успешно, хотя и без результата. Если убрать блок EXCEPTION из функции и запустить процедуру, то произойдет аварийное завершения всей цепочки вызовов подпрограмм. Завершая обсуждение использования EXCEPTION, отметим, что желательно его использовать в местах, где потенциально могут возникать исключения. Более подробное ознакомления с исключениями оставим для самостоятельного изучения.

Уберем дублирующие записи в mytab. После этого удалим одну запись с сохранение ее в переменной.

DO

$$

DECLARE rec record;

BEGIN

DELETE FROM mytab WHERE id = 1 RETURNING \* INTO rec;

RAISE NOTICE 'rec = %', rec;

END;

$$;

Выполним.

ЗАМЕЧАНИЕ: rec = (1,один)

Теперь изменим строку в таблице.

DO

$$

DECLARE rec record;

BEGIN

UPDATE mytab

SET data = data || '!!!'

WHERE id = 20 RETURNING \* INTO rec;

RAISE NOTICE 'rec = %', rec;

END;

$$;

Результат.

ЗАМЕЧАНИЕ: rec = (20,двадцать!!!)

## 2.7. Табличные функции

В PL/pgSQL в объявлении табличной функции необходимо указать, что возвращаются множество строк, используя предложения RETURNS SETOF *тип* или RETURNS TABLE (*тип*). А теле функции возвращаемому запросу должно предшествовать предложение RETURN QUERY *запрос*.

Формировать возвращаемое множество строк можно построчно, например, используя цикл. Для этих целей служит предложение RETURN NEXT *строка.* RETURN NEXT не завершает работу функции, а добавляет в эту строку в возвращаемое множество строк. Результат будет возвращен после завершения функции.

Перепишем SQL функцию get\_seats\_aircraft, которая возвращает места для заданного кода самолета.

CREATE OR REPLACE FUNCTION get\_seats\_aircraft(\_aircraft\_code  
 text)  
 RETURNS SETOF seats STABLE LANGUAGE plpgsql

AS $$

BEGIN

RETURN QUERY SELECT \*

FROM seats

WHERE aircraft\_code = \_aircraft\_code;

END

$$;

Или.

CREATE OR REPLACE FUNCTION get\_seats\_aircraft(\_aircraft\_code text)  
 RETURNS TABLE(aircraft\_code char(3),

seat\_no varchar(4),

fare\_conditions varchar(10))

STABLE LANGUAGE plpgsql

AS $$

BEGIN

RETURN QUERY SELECT \*

FROM seats

WHERE aircraft\_code = \_aircraft\_code;

END

$$;

В следующем примере применим RETURN NEXT, чтобы получить множество текстовых строк.

CREATE OR REPLACE FUNCTION get\_diap( count int )

RETURNS SETOF text

IMMUTABLE LANGUAGE plpgsql

AS $$

BEGIN

FOR i IN 1..count

LOOP

RETURN NEXT to\_char( i, '099' );

END LOOP;

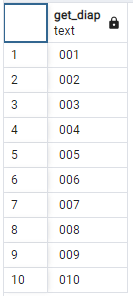
END

$$;

Выполним.

SELECT \* FROM get\_diap( 10);

Результат.



## 2.8. Процедуры.

Процедура не возвращает результат посредством RETURNS, но она может возвращать результат посредством выходных параметров. Процедура может управлять транзакциями, используя COMMIT и ROLLBACK. Процедура вызывается с помощью оператора CALL. При вызове каждому аргументу OUT и INOUT процедуры должна соответствовать переменная. Процедура может вызываться процедурой, анонимным блоком и функцией.

В процедурах при завершении транзакции командами COMMIT и ROLLBACK, новая транзакция будет начата автоматически (в других SQL серверах надо явно начинать новую транзакцию). Ниже приведен пример, взятый из документации на PostgreSQL, иллюстрирующий работу транзакций.

CREATE PROCEDURE transaction\_test1()

LANGUAGE plpgsql

AS $$

BEGIN

FOR i IN 0..9 LOOP

INSERT INTO test1 (a) VALUES (i);

IF i % 2 = 0 THEN

COMMIT;

ELSE

ROLLBACK;

END IF;

END LOOP;

END;

$$;

В этом примере в таблицу test1, состоящую из одного поля a integer, в цикле вставляются записи со значениями от 0 до 9. При i равном нулю в таблицу test1 вставляется строка с 0, после сравнения i % 2 вызывается команда COMMIT, фиксирующая и закрывающая транзакцию и сразу открывает новую транзакцию. Тем самым в таблице сохранена строка. Для i равным 1 после вставки строки с 1 вызывается команда ROLLBACK, откатывающая текущую транзакцию и начинает новую. Строка с 1 отбрасывается. И так далее. В подтверждении сказанного приведем результат работы процедуры (рис. 6):

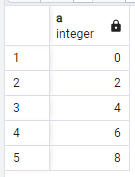


Рисунок 6 – Результат работы transaction\_test1

Обратите внимание, что процедура может возвращать кроме скалярных значений, составные типы, в частности запись таблицы.

CREATE OR REPLACE PROCEDURE get\_seats(IN \_aircraft\_code text,  
 OUT rec seats)

LANGUAGE plpgsql

AS $$

BEGIN

SELECT \* INTO STRICT rec

FROM seats

WHERE aircraft\_code = \_aircraft\_code LIMIT 1;

END;

$$;

Для получения одной записи необходимо использовать уникальный ключ. По коду самолета мы получим множество мест. В примере мы указали STRICT, требующую получения строго одной записи и, если не написать LIMIT 1 (вернуть одну строку), процедура завершится аварийно.

DO

$$

DECLARE

rec seats;

BEGIN

CALL get\_seats('319', rec);

RAISE NOTICE 'rec = %', rec;

END;

$$;

Результа.

ЗАМЕЧАНИЕ: rec = (319,2A,Business)

Забегая вперед, отметим, что для получения множества строк в процедуре надо воспользоваться курсором. Вернемся к нашему примеру с пассажирскими местами. Теперь получим все места для заданного типа самолета.

CREATE OR REPLACE PROCEDURE get\_seats\_tab(  
 IN \_aircraft\_code text,  
 OUT recs refcursor)

LANGUAGE plpgsql

AS $$

BEGIN

OPEN recs FOR

SELECT \*

FROM seats WHERE aircraft\_code = \_aircraft\_code;

END;

$$;

В качестве выходного параметра передается тип курсор. Открываем курсор и связываем его с запросом. Воспользуемся процедурой.

DO

$$

DECLARE

recs refcursor;

rec seats;

BEGIN

CALL get\_seats\_tab('319', recs);

LOOP

FETCH recs INTO rec;

EXIT WHEN NOT FOUND; -- FOUND: выбрана ли очередная строка?

RAISE NOTICE '%', rec;

END LOOP;

CLOSE recs;

END;

$$;

В цикле выполнятся выборка записей из курсора командой FETCH и помещается в переменную rec, идентичную строке таблицы. Когда дойдем до конца FOUND станет равным false и цикл прервется.

# 3. Практическое задание ЛР

Написать хранимые функции и процедуры, выполняющие следующие операции для своей предметной области.

* Добавление одной записи, только через вызов процедуры, для таблиц *Книга, Выдача, Читатель* (на примере варианта №1). Проверять там, где надо, на дублирование записей с выдачей сообщения об ошибке. Проверять на корректность ввода данных, например даты выдачи позже даты возврата.
* Изменение одной записи, только через вызов процедуры, для таблиц *Книга, Выдача, Читатель*. Проверять там, где надо, на дублирование записей с выдачей сообщения об ошибке.
* Удаление одной записи только, через вызов процедуры, для таблиц *Книга, Выдача, Читатель*. Проверять на возможность удаления. Например, удаляем книгу, которая находится на руках у читателя (не допускать формирование исключения о нарушении целостности БД).
* Запрос читателей по первичному ключу, по адресу проживания, по выданной книге.
* Запрос списка книг, выданных заданному читателю.

# 4. Темы для самостоятельной проработки

* Функции на языке запросов (SQL)

<https://postgrespro.ru/docs/postgrespro/11/xfunc-sql>

* Перегрузка функций

https://postgrespro.ru/docs/postgrespro/11/xfunc-overload

* Категории изменчивости функций

https://postgrespro.ru/docs/postgrespro/11/xfunc-volatility

* Сообщения и ошибки

https://postgrespro.ru/docs/postgrespro/11/plpgsql-errors-and-messages

* Составные типы

<https://postgrespro.ru/docs/postgrespro/11/rowtypes>

* Процедурный язык PL/pgSQL.

[https://postgrespro.ru/docs/postgrespro/11/plpgsql](https://postgrespro.ru/docs/postgrespro/11/plpgsql%20)

# 5. Примеры вопросов для самостоятельной проработки

* Рассказать для чего нужны функции и процедуры.
* Чем отличается функция от процедуры.
* Рассказать об аргументах функции и процедуры.
* Рассказать о типах возвращаемых параметров.
* Что такое категории изменчивости?
* Рассказать о перегрузке подпрограмм.
* Рассказать о полиморфизме.
* Что такое составные типы? Как обращаться к атрибутам составного типа?
* Как функции вернуть запись таблицы и результат работы SELECT?
* Что такое табличные функции? Привести пример использования.
* Что такое анонимный блок в PL/pgSQL?
* Рассказать об условных операторах.
* Рассказать о циклах.
* Чем отличаются функции, написанные на SQL, от функций, написанных на PL/pgSQL?
* Чем отличаются процедуры, написанные на SQL, от процедур, написанных на PL/pgSQL?
* Рассказать о командах, возвращающих одну строку. Их использовании в подпрограммах.
* Рассказать о табличные функциях в PL/pgSQL.
* Рассказать об исключениях.

# 6. СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Фомин М.М. Реляционные базы данных. Учебное пособие для бакалавров <https://e-learning.bmstu.ru/iu6/mod/resource/view.php?id=6634>
2. Карпова И.П. Базы данных. Учебное пособие. – Московский государственный институт электроники и математики (Технический университет): учебное пособие– М., 2009. – 140-141 с, 102 c.
3. PostgreSQL. Основы языка SQL: учеб. пособие / Е. П. Моргунов; под ред. Е. В. Рогова, П. В. Лузанова. - СПб.: БХВ-Петербург, 2018. - 65-68 c, 68-72 с.
4. Стоунз Р., Мэтью Н. PostgreSQL. Основы. – Пер. с англ. – СПб: Символ\_Плюс, 2002. – 640 с., ил. ISBN 5\_93286\_043\_X
5. Болье, А. Изучаем SQL. Генерация, выборка и обработка данных, 3-е изд./ пер. с англ. И.В. Красикова. — Киев.: “Диалектика”,2021. — 402 с.: ил. ISBN 978-1-492-05761-1