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)



**МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ**

**ПО КУРСУ «БАЗЫ ДАННЫХ»**

# Лабораторная работа №4 «Массивы, курсоры, триггеры, роли»

Авторы:

Кудрявцев А.П.

Фомин М.М.

Скворцова М.А.

Лапшин А.В.

Москва, 2023

# 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 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 №4. Массивы, курсоры, триггеры, роли

Данная лабораторная состоит из четырех взаимосвязанных частей и является продолжением лабораторной работы №3. В каждом разделе есть теоретическая и практическая части. В конце есть задания для самостоятельного выполнения в рамках лабораторной работы.

**Цель.**

Данная лабораторная работа призвана сформировать у студента понимание назначения массивов, курсоров, триггеров и ролей, их написание и использование.

**Задачи:**

* Ознакомиться с использованием массивов.
* Научиться (изменять\добавлять\удалять) данные в массиве с помощью встроенных операций.
* Получить знания о курсорах и научиться использовать курсоры.
* Узнать о ролях и пользователях.
* Научиться ыпользоваться командами GRANT и REVOKE для того, чтобы (предоставлять\отзывать) доступ к данным.

# 1. Массивы

Массив – это тип данных, в котором хранится упорядоченный набор однотипных элементов. Обращаться к элементу массива можно по целочисленному индексу в квадратных скобках.

*массив*[*индекс*]

Можно извлечь несколько соседних элементов («срез массива»).

*массив*[*нижний\_индекс*:*верхний\_индекс*]

По умолчанию, нумерация элементов массива начинается с 1. Массивы могут быть многомерными, но они должны прямоугольными.

Для того, чтобы объявить массив нужно добавить квадратные скобки к типу.

names text[];

dates date[];

a int[][];

Указывать размер массива в квадратных скобках не надо, т.к. он игнорируется. PostgreSQL не накладывает ограничения на размер массива.

 Инициализировать массив можно несколькими способами.

По-элементная инициализация.

DO $$

DECLARE

 arr int[];

BEGIN

 arr[3] = 3;

 arr[5] = 5;

 arr[6] = 6;

 FOR i IN 1..6 LOOP

 RAISE NOTICE 'arr[%]=%', i, arr[i];

 END LOOP;

END;

$$;

Результат.

ЗАМЕЧАНИЕ: arr[1]=<NULL>

ЗАМЕЧАНИЕ: arr[2]=<NULL>

ЗАМЕЧАНИЕ: arr[3]=3

ЗАМЕЧАНИЕ: arr[4]=<NULL>

ЗАМЕЧАНИЕ: arr[5]=5

ЗАМЕЧАНИЕ: arr[6]=6

Как видим, не инициализированные элементы массива не определены. В связи с этим возникает вопрос, а какой реальный размер массива и с какого индекса начинается отсчет элементов. Для этого воспользуемся встроенными функциями:

* array\_length(), определяющую длину массива;
* array\_lower(), нижняя граница индекса массива;
* array\_upper(), верхняя граница индекса массива.

DO $$

DECLARE

 arr int[];

BEGIN

 arr[3] = 3;

 arr[5] = 5;

 arr[6] = 6;

 RAISE NOTICE 'длина arr[]=%', array\_length(arr, 1 );

 FOR i IN array\_lower(arr, 1)..array\_upper(arr, 1) LOOP

 RAISE NOTICE 'arr[%]=%', i, arr[i];

 END LOOP;

END;

$$;

Результат.

ЗАМЕЧАНИЕ: длина arr[]=4

ЗАМЕЧАНИЕ: arr[3]=3

ЗАМЕЧАНИЕ: arr[4]=<NULL>

ЗАМЕЧАНИЕ: arr[5]=5

ЗАМЕЧАНИЕ: arr[6]=6

При желании, мы можем заполнять массив, начиная с любого индекса.

Инициализация с помощью конструктора массива. Конструктор простого массива состоит из ключевого слова ARRAY.

names text[] = ARRAY[‘Николай’, ‘Андрей’, ‘Иван’, ‘Сергей’];

dates date[] = ARRAY['15.01.2023', '20.02.2023','30.03.2023'];

a int[][] = ARRAY[ARRAY[10, 11, 12], ARRAY[13, 14, 15]];

b int[][] = ARRAY[[10, 11, 12], [13, 14, 15]];

Применим.

DO $$

DECLARE

 arr varchar(10)[] = ARRAY['один', 'два','три'];

BEGIN

 FOR i IN 1..3 LOOP

 RAISE NOTICE 'arr[%]=%', i, arr[i];

 END LOOP;

END;

$$;

Результат.

ЗАМЕЧАНИЕ: arr[1]=один

ЗАМЕЧАНИЕ: arr[2]=два

ЗАМЕЧАНИЕ: arr[3]=три

Также возможно создать массив из результатов подзапроса. В этом случае конструктор массива записывается так же с ключевым словом ARRAY, за которым в круглых скобках следует подзапрос. Подзапрос должен возвращать один столбец.

DO $$

DECLARE

 codes char(3)[] = ARRAY(SELECT aircraft\_code FROM aircrafts);

BEGIN

 FOR i IN 1..array\_upper(codes, 1) LOOP

 RAISE NOTICE 'codes[%]=%', i, codes[i];

 END LOOP;

END;

$$;

Результат.

ЗАМЕЧАНИЕ: codes[1]=773

ЗАМЕЧАНИЕ: codes[2]=763

ЗАМЕЧАНИЕ: codes[3]=SU9

ЗАМЕЧАНИЕ: codes[4]=321

ЗАМЕЧАНИЕ: codes[5]=733

ЗАМЕЧАНИЕ: codes[6]=CN1

ЗАМЕЧАНИЕ: codes[7]=CR2

ЗАМЕЧАНИЕ: codes[8]=320

ЗАМЕЧАНИЕ: codes[9]=319

Приведем пример использования двумерного массива.

DO $$

DECLARE

 arr text[][] = '{{"один","два"}, {"десять", "одиннадцать"}}';

BEGIN

 FOR i IN 1..array\_upper(arr, 1) LOOP

 FOR j IN 1..array\_upper(arr, 2) LOOP

 RAISE NOTICE 'arr[%][%]=%', i, j, arr[i][j];

 END LOOP;

 END LOOP;

END;

$$;

Результат.

ЗАМЕЧАНИЕ: arr[1][1]=один

ЗАМЕЧАНИЕ: arr[1][2]=два

ЗАМЕЧАНИЕ: arr[2][1]=десять

ЗАМЕЧАНИЕ: arr[2][2]=одиннадцать

Надо отметь, что после инициализации многомерного массива нельзя изменять его размер.

Для массива есть специальный цикл FOREACH, в котором происходит перебор элементов массива. Синтаксис цикла FOREACH:

[<<*метка*>>]

FOREACH *цель* [SLICE *число*] IN ARRAY *выражение* LOOP

*операторы*

END LOOP [*метка*];

Без указания SLICE, или если SLICE равен 0, цикл выполняется по всем элементам массива, полученного из *выражения*. Переменной *цель* последовательно присваивается каждый элемент массива и для него выполняется тело цикла.

 Пример цикла по элементам двумерного массива.

DO $$

DECLARE

 arr text[][] = ARRAY[ARRAY['один','два'],

 ARRAY['десять', 'одиннадцать'],

 ARRAY['двадцать', 'двадцать один']];

 item text;

BEGIN

 FOREACH item IN ARRAY arr LOOP

 RAISE NOTICE 'item=%', item;

 END LOOP;

END;

$$;

Результат.

ЗАМЕЧАНИЕ: item=один

ЗАМЕЧАНИЕ: item=два

ЗАМЕЧАНИЕ: item=десять

ЗАМЕЧАНИЕ: item=одиннадцать

ЗАМЕЧАНИЕ: item=двадцать

ЗАМЕЧАНИЕ: item=двадцать один

При положительном значении SLICE выполняет итерации по срезам массива, а не по отдельным элементам. Значение SLICE должно быть целым числом, не превышающим размерности массива. Переменная *цель* должна быть массивом, который получает последовательные срезы исходного массива, где размерность каждого среза задаётся значением SLICE.

DO $$

DECLARE

 arr text[][] = ARRAY[ARRAY['один','два'],

 ARRAY['десять', 'одиннадцать'],

 ARRAY['двадцать', 'двадцать один']];

 row text[];

BEGIN

 FOREACH row SLICE 1 IN ARRAY arr LOOP

 RAISE NOTICE 'row=%', row;

 END LOOP;

END;

$$;

Результат.

ЗАМЕЧАНИЕ: row={один,два}

ЗАМЕЧАНИЕ: row={десять,одиннадцать}

ЗАМЕЧАНИЕ: row={двадцать,"двадцать один"}

PostgreSQL позволяет создавать в таблицах столбцы, в которых содержатся массивы. Эти массивы могут быть многомерными и могут содержать значения любого из встроенных типов, а также типов данных, определенных пользователем.

Предположим, что нам необходимо сформировать и сохранить в базе данных в удобной форме графики работы пилотов авиакомпании, т. е. номера дней недели, когда они совершают полеты.

Создадим таблицу, в которой эти графики будут храниться в виде одномерных массивов.

|  |
| --- |
| CREATE TABLE pilots( pilot\_name text, schedule integer[]);  |

Для указания на то, что это массив, нужно добавить квадратные скобки к наименованию типа данных. При этом задавать число элементов не обязательно. Заполним таблицу четырьмя записями.

Массив в команде вставки представлен в виде строкового литерала с указанием типа данных и квадратных скобок, означающих массив. Обратите внимание, что все массивы имеют различное число элементов.

|  |
| --- |
| INSERT INTO pilotsVALUES (‘Ivan', ‘{1, 3, 5, 6, 7}'::integer[] ),(‘Petr', ‘{1, 2, 5, 7}'::integer[] ),(‘Pavel', ‘{2, 5}'::integer[] ),(‘Boris', ‘{3, 5, 6}'::integer[] );  |

Предположим, что руководство компании решило, что каждый пилот должен летать 4 раза в неделю. Значит, нам придется обновить значения в таблице. Пилоту по имени Boris добавим один день с помощью операции конкатенации:

UPDATE pilots SET schedule = schedule || 7 WHERE pilot\_name = 'Boris';

Пилоту по имени Pavel добавим один день в конец массива с помощью функции array\_append:

UPDATE pilots

 SET schedule = array\_append( schedule, 6 )

WHERE pilot\_name = 'Pavel';

Ему же добавим один день в начало списка с помощью функции array\_prepend (обратите внимание, что параметры функции поменялись местами):

UPDATE pilots

 SET schedule = array\_prepend( 1, schedule )

WHERE pilot\_name = 'Pavel';

У пилота по имени Ivan имеется лишний день в графике. С помощью функции array\_remove удалим из графика пятницу (второй параметр функции указывает значение элемента массива, а не индекс):

UPDATE pilots

 SET schedule = array\_remove( schedule, 5 )

WHERE pilot\_name = 'Ivan';

У пилота по имени Petr изменим дни полетов, не изменяя их общего количества. Воспользуемся индексами для работы на уровне отдельных элементов массива. По умолчанию нумерация индексов начинается с единицы, а не с нуля. При необходимости ее можно изменить. К элементам одного и того же массива можно обращаться в предложении SET по отдельности, как будто это разные столбцы.

UPDATE pilots

 SET schedule[1] = 2, schedule[2] = 3

WHERE pilot\_name = 'Petr';

А можно было бы, используя срез (slice) массива, сделать и так:

UPDATE pilots

 SET schedule[ 1:2 ] = ARRAY[ 2, 3 ]

WHERE pilot\_name = 'Petr';

Теперь продемонстрируем основные операции, которые можно применять к массивам, выполняя выборки из таблиц. Получим список пилотов, летающих по средам:

SELECT \*

FROM pilots

WHERE array\_position( schedule, 3 ) IS NOT NULL;

Функция array\_position возвращает индекс первого вхождения элемента с указанным значением в массив. Если же такого элемента нет, она возвратит NULL. Выберем пилотов, летающих по понедельникам и воскресеньям:

SELECT \*

FROM pilots

WHERE schedule @> '{ 1, 7 }'::integer[];

Оператор @> означает проверку того факта, что в левом массиве содержатся все элементы правого массива. Конечно, при этом в левом массиве могут находиться и другие элементы, что мы и видим в графике этого пилота. Еще аналогичный вопрос: кто летает по вторникам и/или по пятницам? Для получения ответа воспользуемся оператором &&, который проверяет наличие общих элементов у массивов, т. е. пересекаются ли их множества значений. В нашем примере число общих элементов, если они есть, может быть равно одному или двум. Здесь мы использовали нотацию с ключевым словом ARRAY, а не '{2, 5}'::integer[]. Так же можно применять ту, которая принята в рамках выполнения вашего проекта.

SELECT \*

FROM pilots

WHERE schedule && ARRAY[ 2, 5 ];

Сформулируем вопрос в форме отрицания: кто не летает ни во вторник, ни в пятницу? Для получения ответа добавим в предыдущую SQL-команду отрицание NOT:

SELECT \*

FROM pilots

WHERE NOT ( schedule && ARRAY[ 2, 5 ] );

Иногда требуется развернуть массив в виде столбца таблицы. В таком случае поможет функция unnest:

SELECT unnest( schedule ) AS days\_of\_week

FROM pilots

WHERE pilot\_name = 'Ivan';

# 2. Курсоры

SQL операции позволяют работать с множеством строк. Однако, это может занимать много места, так чтобы избежать переполнение памяти надо обрабатывать результат по частям, используя курсор. Еще одна из причин использования курсора — это тогда, когда результатом запроса является не тривиальный анализ и обработка нескольких таблиц (результат которых помещается во временную таблицу).

## 2.1. Объявление курсора

Объявить курсор в PL/pgSQL можно двумя способами. Первый, объявить курсорную переменную, которая будет связана с запросом позже.

*имя* refcursor;

Второй способ, имеет следующий синтаксис.

*имя* [[NO] SCROLL] CURSOR [(*аргументы*)] FOR *запрос*;

С указанием SCROLL курсор можно будет прокручивать назад. При NO SCROLL прокрутка назад не разрешается. Если ничего не указано, то возможность прокрутки назад зависит от запроса. Если указаны *аргументы*, то они должны представлять собой пары *имя тип\_данных*, разделённые через запятую. Эти пары определяют имена, которые будут заменены значениями параметров в данном запросе. Фактические значения для замены этих имён появятся позже, при открытии курсора.

## 2.2. Открытие курсора

После объявления курсора перед использованием его необходимо открыть. Курсор закрывается автоматически в конце транзакции. Но, если необходимо использовать курсор повторно или освободить ресурсы, то его нужно закрыть (CLOSE *курсор*).

Открытие не связанной с запросом курсорной переменной.

*OPEN несвязанная\_переменная\_курсора [[NO] SCROLL] FOR запрос*;

Запрос должен быть командой SELECT или любой другой, которая возвращает строки.

 Пример.

DO $$

DECLARE

 cur refcursor;

BEGIN

 OPEN cur FOR SELECT \* FROM aircrafts;

END;

$$;

Открытие связанного курсора.

*OPEN связанная\_переменная\_курсора [([аргумент] [, ...])];*

Пример.

DO $$

DECLARE

 cur CURSOR(\_range int) FOR SELECT \*

 FROM aircrafts

 WHERE \_range <= range;

BEGIN

 OPEN cur(3000);

END;

$$;

## 2.3. Операции с курсором

После открытия курсора выполняется позиционирование на позицию «выше» первой строки курсора. Далее, использование команд, работающих с курсором, позволит нам перебирать строки и вносить в них изменения и сохранять в БД.

### 2.3.1. Извлечение текущей строки

Для получения строки курсора используется следующая команда.

*FETCH [направление {FROM | IN}] курсор INTO цель;*

В качестве *цели* может быть переменная-кортеж, переменная типа record или разделённый запятыми список простых переменных. Если строки нет (т.е. текущая позиция «ниже» последней строки или «выше» первой), *цели* присваивается NULL. Проверить, была ли получена запись, можно при помощи специальной переменной FOUND.

Здесь *направление* может принимать следующие значения.

* NEXT – Выбрать следующую строку. Это действие подразумевается по умолчанию, если *направление* опущено.
* PRIOR – Выбрать предыдущую строку.
* FIRST – Выбрать первую строку запроса (аналогично указанию ABSOLUTE 1).
* LAST – Выбрать последнюю строку запроса
(аналогично ABSOLUTE -1).
* ABSOLUTE *число* – Выбрать строку под номером *число* с начала, либо под номером abs(*число*) с конца, если *число* отрицательно. Если *число* выходит за границы набора строк, курсор размещается перед первой или после последней строки; в частности, с ABSOLUTE 0 курсор оказывается перед первой строкой.
* RELATIVE *число* – Выбрать строку под номером *число*, считая со следующей вперёд, либо под номеро мabs(*число*), считая с предыдущей назад, если *число* отрицательно. RELATIVE 0 повторно считывает текущую строку, если таковая имеется.

Значения *направления*, которые требуют перемещения назад, приведут к ошибке, если курсор не был объявлен или открыт с указанием SCROLL.

В стандарте SQL перед именем курсора допускается только указание FROM; возможность указать IN или опустить оба указания относится к расширениям.

Пример.

DO $$

DECLARE

 rec record;

 cur CURSOR(\_range int) FOR SELECT \*

 FROM aircrafts

 WHERE \_range <= range;

BEGIN

 OPEN cur(3000);

 LOOP

 FETCH cur INTO rec;

 EXIT WHEN NOT FOUND;

 RAISE NOTICE 'rec=%', rec;

 END LOOP;

END;

$$;

Результат.

ЗАМЕЧАНИЕ: rec=(773,"Boeing 777-300",5000)

ЗАМЕЧАНИЕ: rec=(763,"Boeing 767-300",7900)

ЗАМЕЧАНИЕ: rec=(SU9,"Sukhoi SuperJet-100",3000)

ЗАМЕЧАНИЕ: rec=(321,"Airbus A321-200",5600)

ЗАМЕЧАНИЕ: rec=(733,"Boeing 737-300",4200)

ЗАМЕЧАНИЕ: rec=(320,"Airbus A320-200",6033)

ЗАМЕЧАНИЕ: rec=(319,"Airbus A319-100",6930)

### 2.3.2. Цикл FOR

Можно использовать цикл FOR. В цикле последовательно (если мы не вмешаемся, вызвав, к примеру, MOVE cur) считываются данные и помещаются в переменную цикла.

DO $$

DECLARE

 rec record;

 cur CURSOR(\_range int) FOR SELECT \*

 FROM aircrafts

 WHERE \_range <= range;

BEGIN

 FOR rec IN cur(3000) LOOP

 RAISE NOTICE 'rec=%', rec;

 END LOOP;

END;

$$;

Следует обратить внимание на то, что курсор явно не открывается, его открывает цикл FOR.

Рассмотрим еще один вариант задание курсора в цикле. Запрос задаем

DO $$

DECLARE

 code text;

BEGIN

 FOR code IN SELECT aircraft\_code FROM aircrafts

 WHERE 3000 <= range LOOP

 RAISE NOTICE 'aircraft\_code=%', code;

 END LOOP;

END;

$$;

в объявлении цикла. Для разнообразия в выборку делаем по одному полю.

### 2.3.3. Перемещение курсора без извлечения данных

Перемещение курсора без извлечения данных выполнятся командой MOVE.

MOVE [*направление* {FROM | IN}] *курсор*;

MOVE работает точно так же, как и FETCH, но при этом только перемещает курсор и не извлекает строку, к которой переместился. Параметры MOVE так же совпадают с параметрами FETCH.

DO $$

DECLARE

 rec record;

 cur CURSOR(\_range int) FOR SELECT \*

 FROM aircrafts

 WHERE \_range <= range;

BEGIN

 FOR rec IN cur(3000) LOOP

 RAISE NOTICE 'rec=%', rec;

 MOVE cur;

 END LOOP;

END;

$$;

Результат.

ЗАМЕЧАНИЕ: rec=(773,"Boeing 777-300",5000)

ЗАМЕЧАНИЕ: rec=(SU9,"Sukhoi SuperJet-100",3000)

ЗАМЕЧАНИЕ: rec=(733,"Boeing 737-300",4200)

ЗАМЕЧАНИЕ: rec=(319,"Airbus A319-100",6930)

### 2.3.4. Изменение и удаление строк

Когда курсор связан с одной таблицей, то текущую строку можно изменить или удалить. Сослаться на текущую строку в UPDATE и DELETE можно с помощью CURRENT OF.

UPDATE *таблица* SET ... WHERE CURRENT OF *курсор*;

DELETE FROM *таблица* WHERE CURRENT OF *курсор*;

Пример.

DO $$

DECLARE

 rec record;

 cur CURSOR(\_range int) FOR SELECT \*

 FROM aircrafts

 WHERE \_range <= range;

BEGIN

 FOR rec IN cur(3000) LOOP

 UPDATE aircrafts

 SET range = range+1

 WHERE CURRENT OF cur;

 END LOOP;

END;

$$;

## 2.4. Возврат курсора из функции

Вместо того, чтобы сразу выполнять весь запрос, есть возможность настроить курсор, инкапсулирующий запрос, и затем получать результат запроса по нескольку строк за раз. Одна из причин так делать заключается в том, чтобы избежать переполнения памяти, когда результат содержит большое количество строк. Более интересным вариантом использования является возврат из функции ссылки на курсор, что позволяет вызывающему получать строки запроса. Это эффективный способ получать большие наборы строк из функций.

Работать с курсором необязательно в той же функции, где он был открыт. Из функции можно вернуть значение с типом refcursor, что позволит вызывающему продолжить работу с курсором. (Внутри refcursor представляет собой обычное строковое имя так называемого портала, содержащего активный запрос курсора. Это имя можно передавать, присваивать другим переменным с типом refcursor и так далее, при этом портал не нарушается.)

Для PL/pgSQL использовать курсор, открытый функцией, достаточно просто. В примере открываем курсор и его возвращаем.

CREATE FUNCTION get\_aircrafts\_cur(\_range int) RETURNS refcursor

 LANGUAGE plpgsql

AS $$

DECLARE

 cur CURSOR( rng int) FOR SELECT \*

 FROM aircrafts

 WHERE rng <= range

 ORDER BY aircraft\_code;

BEGIN

 OPEN cur(\_range);

 RETURN cur;

END;

$$;

Вызовем функцию и распечатаем данные.

DO $$

DECLARE

 cur refcursor;

 rec record;

BEGIN

 cur = get\_aircrafts\_cur(3000);

 LOOP

 FETCH cur INTO rec;

 EXIT WHEN NOT FOUND;

 RAISE NOTICE 'rec=%', rec;

 END LOOP;

END;

$$;

Результат.

ЗАМЕЧАНИЕ: rec=(319,"Airbus A319-100",6930)

ЗАМЕЧАНИЕ: rec=(320,"Airbus A320-200",6033)

ЗАМЕЧАНИЕ: rec=(321,"Airbus A321-200",5600)

ЗАМЕЧАНИЕ: rec=(733,"Boeing 737-300",4200)

ЗАМЕЧАНИЕ: rec=(763,"Boeing 767-300",7900)

ЗАМЕЧАНИЕ: rec=(773,"Boeing 777-300",5000)

ЗАМЕЧАНИЕ: rec=(SU9,"Sukhoi SuperJet-100",3000)

Использование курсора, сформированного функцией в SQL напрямую невозможно. Однако, можно обращаться к курсору по имени, представляющему собой строку (если имя курсору не присвоено, то система присвоит какое-то имя). Один из способов, это передать функции имя курсора через аргумент функции.

CREATE FUNCTION get\_aircrafts\_cur2(\_range int, cur refcursor)
 RETURNS refcursor LANGUAGE plpgsql

AS $$

DECLARE

BEGIN

 OPEN cur FOR SELECT \*

 FROM aircrafts

 WHERE \_range <= range

 ORDER BY aircraft\_code;

 RETURN cur;

END;

$$;

Вызовем.

SET search\_path = bookings;

SELECT get\_aircrafts\_cur2(3000, 'my\_cur');

FETCH ALL IN "my\_cur";

Результат.



Наиболее удобный способ — это присвоить курсору имя функции, избегая перегрузку этой функций.

CREATE FUNCTION get\_aircrafts\_cur(\_range int) RETURNS refcursor

 LANGUAGE plpgsql

AS $$

DECLARE

 cur CURSOR( rng int) FOR SELECT \*

 FROM aircrafts

 WHERE rng <= range

 ORDER BY aircraft\_code;

BEGIN

 cur = 'get\_aircrafts\_cur';

 OPEN cur(\_range);

 RETURN cur;

END;

$$;

Применим.

SELECT get\_aircrafts\_cur(3000);

FETCH ALL IN "get\_aircrafts\_cur";

Результат.



# 3. Триггеры

## 3.1. Общие сведения

Триггеры – это программный объект БД, который выполняется на стороне сервера. Во многих случаях это позволяет повысить производительность системы, уменьшить сетевой трафик. Напрямую обратиться к триггеру невозможно. Он вызывается автоматически при наступлении соответствующего события БД – добавления новой строки в таблицу, изменении или удалении строки. Триггер может сработать, когда соответствующие действия над БД выполняет клиентское приложение, хранимая подпрограмма или триггер (другой или тот же самый).

Триггеры выполняются в контексте той транзакции, под управлением которой работает программа, инициализирующая вызов триггера.

В PostgreSQL можно создавать триггеры манипулирования данными (DML) и триггеры событий (DDL). Здесь рассмотрим только триггеры DML, которые вызываются при выполнении операторов INSERT, UPDATE, DELETE и TRUNCATE. Можно создать два типа триггеров – *операторные* (STATEMENT) и *построчные* (ROW) триггеры. Первые вызываются при выполнении оператора SQL – INSERT, UPDATE, DELETE (изменения могут затронуть одну или более строк), вторые – для каждой строки, которая должна быть изменена.

## 3.2. События

Триггеры устанавливаются на таблицы и представления, которые срабатывают при выполнении SQL команд (INSERT, UPDATE, DELETE). Триггеры различаются по времени вызова: до начала выполнения операции – BEFOR и после – AFTER. Но перед тем, как рассмотреть все возможные комбинации создания триггеров, рассмотрим синтаксис триггера. Ниже приведено упрощенное описание триггера, которое соответствует рамкам курса.

CREATE [OR REPLACE] TRIGGER *имя* {BEFORE | AFTER | INSTEAD OF}

{INSERT | UPDATE [ OF *имя\_столбца* [, ...]] | DELETE | TRUNCATE}

ON *имя\_таблицы*

[REFERENCING {{OLD|NEW} TABLE *имя\_переходного\_отношения*} [...]]

[FOR [EACH] {ROW | STATEMENT}]

[WHEN (*условие*)]

EXECUTE FUNCTION *имя\_функции* (*аргументы*);

Триггер описывает условие, когда должна быть вызвана функция, заданная предложением EXECUTE FUNCTION *имя\_функции* (*аргументы*). Триггерная функция — это обычная функция без аргументов и должна возвращать тип trigger. Аргументы функции в объявлении триггера передаются функции через специальную переменную массив TG\_ARGV[], но об этом позже. Итак, функция имеет вид.

CREATE FUNCTION *имя\_функции*() RETURNS trigger

LANGUAGE plpgsql

AS $$

DECLARE

…

BEGIN

…

END;

$$;

Когда функция на PL/pgSQL срабатывает как триггер, в блоке верхнего уровня автоматически создаются несколько специальных переменных:

* NEW — тип данных record. Переменная содержит новую строку базы данных для команд INSERT/UPDATE в триггерах уровня строки. В триггерах уровня оператора и для команды DELETE эта переменная имеет значение NULL.
* OLD — тип данных record. Переменная содержит старую строку базы данных для команд UPDATE/DELETE в триггерах уровня строки. В триггерах уровня оператора и для команды INSERT эта переменная имеет значение NULL.
* TG\_NAME — тип данных name. Переменная содержит имя сработавшего триггера.
* TG\_WHEN — тип данных text. Строка, содержащая BEFORE, AFTER или INSTEAD OF, в зависимости от определения триггера.
* TG\_LEVEL — тип данных text. Строка, содержащая ROW или STATEMENT, в зависимости от определения триггера.
* TG\_OP — тип данных text. Строка, содержащая INSERT, UPDATE, DELETE или TRUNCATE, в зависимости от того, для какой операции сработал триггер.
* TG\_RELID — тип данных oid (идентификаторы объектов, используются как ключи для системных таблиц). OID таблицы, для которой сработал триггер.
* TG\_TABLE\_NAME — тип данных name. Имя таблицы, для которой сработал триггер.
* TG\_TABLE\_SCHEMA — Тип данных name. Имя схемы, содержащей таблицу, для которой сработал триггер.
* TG\_NARGS — тип данных integer. Число аргументов в команде CREATE TRIGGER, которые передаются в триггерную функцию.
* TG\_ARGV[] — тип данных массив text. Аргументы от оператора CREATE TRIGGER. Индекс массива начинается с 0. Для недопустимых значений индекса (меньше нуля или больше и равно TG\_NARGS) возвращается NULL.

В таблице перечисляются типы триггеров.

|  |  |  |  |
| --- | --- | --- | --- |
| **Когда** | **Событие** | **Построчный** | **Операторный** |
| BEFORE | INSERT/UPDATE/DELETE | Таблицы | Таблицы, представления |
| TRUNCATE | — | Таблицы |
| AFTER | INSERT/UPDATE/DELETE | Таблицы | Таблицы, представления |
| TRUNCATE | — | Таблицы |
| INSTEAD OF | INSERT/UPDATE/DELETE | Представления | — |
| TRUNCATE | — | — |

Приведем порядок срабатывания триггеров. Если для какого-то события задано несколько триггеров, то они выполняются в алфавитном порядке.

1. Операторный триггеры BEFOR. Выполняется один раз до операции.
2. Построчный триггер BEFOR. Выполняется для каждой строки до сохранения изменений в БД.
3. Построчный триггер INSTEAD OF (для представления). Выполняется для каждой строки вместо операции.
4. Построчный триггер AFTER. Выполняется для каждой строки после операции.
5. Операторный триггер AFTER. Выполняется один раз после операции.

В триггере может присутствовать условие его срабатывания WHEN(условие). Если задано условие, то функция будет срабатывать, когда условие возвращает true. Целесообразно использовать WHEN в построчных триггерах, проверяя старые (OLD) и новые (NEW) значения в строках.

В триггере BEFORE условие WHEN вычисляется непосредственно перед возможным вызовом функции, поэтому проверка WHEN существенно не отличается от проверки того же условия в начале функции триггера.

В триггере AFTER условие WHEN проверяется сразу после изменения строки, и, если оно выполняется, событие запоминается, чтобы вызвать триггер в конце оператора. Если же для триггера AFTER условие WHEN не выполняется, нет необходимости запоминать событие для последующей обработки или заново перечитывать строку в конце оператора. Это приводит к значительному ускорению операторов, изменяющих множество строк, когда триггер должен срабатывать только для некоторых из них.

### 3.2.1. Операторный триггер Before

Операторный триггер срабатывает перед началом выполнения команды. Этот триггер вызывается один раз, независимо от количества затронутых строк.

Триггер задается с помощью ключевых слов BEFORE и FOR EACH STATEMENT.

CREATE TRIGGER tr\_foo1

**BEFORE** INSERT OR UPDATE OR DELETE – м.б. любая комбинация

ON mytable

**FOR EACH STATEMENT**

EXECUTE FUNCTION fun\_foo1();

Триггерная функция.

CREATE FUNCTION fun\_foo1() RETURNS trigger

LANGUAGE plpgsql

AS $$

DECLARE

…

BEGIN

…

 **RETURN NULL**;

END;

$$;

Возвращаемое значение триггерной функции должно быть NULL. Триггерная функция с помощью TG переменных может определить контекст вызова:

TG\_WHEN = ‘BEFORE’,

TG\_LEVEL = ‘STATEMENT’,

TG\_OP = ‘INSERT’, ‘UPDATE’ или ‘DELETE’,

…

### 3.2.2. Построчный триггер Before

Построчный триггер BEFORE срабатывает для каждой строки перед тем, как изменения сохранятся в журнале предзаписи. Триггерной функции доступны строка до внесения изменений (OLD) и строка после внесения изменений (NEW). Чтобы данные сохранились в строке, функция должна вернуть строку NEW для вставки и обновлении, для удаления – строку OLD. Триггерная функция может изменить строку NEW. Данная возможность широко используется, но при этом усложняет сопровождение клиентской части программы.

Для отмены действий над строкой следует вернуть NULL. В этом случае, операция будет продолжена, но изменения, связанные с этой строкой, не будут внесены в таблицу.

Триггер задается с помощью ключевых слов BEFORE и FOR EACH ROW.

CREATE TRIGGER tr\_foo2

**BEFORE** INSERT OR UPDATE OR DELETE – м.б. любая комбинация

ON mytable

**FOR EACH ROW**

EXECUTE FUNCTION fun\_foo2();

Триггерная функция.

CREATE FUNCTION fun\_foo2() RETURNS trigger

LANGUAGE plpgsql

AS $$

DECLARE

…

BEGIN

…

 **RETURN NEW/OLD**; (для INSERT, UPDATE/для DELETE)

 или

 **RETURN NULL**; отмена

END;

$$;

Триггерная функция с помощью TG переменных может определить контекст вызова:

OLD – исходная строка до операции (не определена для вставки),

NEW – измененная строка (не определена при удалении),

TG\_WHEN = ‘BEFORE’,

TG\_LEVEL = ‘ROW’,

TG\_OP = ‘INSERT’, ‘UPDATE’ или ‘DELETE’,

…

### 3.2.3. Построчный триггер INSTEAD OF

Построчный триггер INSTEAD OF срабатывает для каждой строки представления и выполняется вместо операции. Это позволяет модифицировать данные представления, состоящего из нескольких таблиц. При наличии триггера INSTEAD OF можно использовать операции INSERT, UPDATE и DELETE. Функция триггера вносит соответствующие изменения в базовые таблицы представления. Если триггер не задан, то операция завершится ошибкой.

При INSERT, UPDATE возвращать необходимо строку NEW, при DELETE – строку OLD. Для отмены – NULL.

Трудно представить ситуацию, когда редактирование данных необходимо выполнять, используя представления. Тем более, что триггер INSTEAD OF выполняет для каждой базовой таблицы соответствующие операции.

### 3.2.4. Построчный триггер AFTER

Построчный триггер AFTER срабатывает для каждой строки после выполнения операции (т.е. все изменения сохранены в журнале предзаписи, транзакция еще не завершилась). Возвращаемое значение игнорируется.

Для триггеров AFTER имеется возможность получить доступ к *переходным таблицам*, содержащим все строки, которые были добавлены, удалены или изменены текущим оператором. Это позволяет триггеру наблюдать общую картину того, что сделал оператор, а не только одну строку за другой. Таблица, содержащая не модифицированные строки и удаленные, задающаяся предложением OLD TABLE, может быть задана только для триггеров, обрабатывающих UPDATE или DELETE. Кроме того, у триггера UPDATE не должно быть список столбцов. Таблица, заданная NEW TABLE, содержит строки модифицированные и вставленные.

Триггер задается с помощью ключевых слов AFTER и FOR EACH ROW.

CREATE TRIGGER tr\_foo3

**AFTER** INSERT OR UPDATE OR DELETE – м.б. любая комбинация

**REFERENCING**

 **NEW TABLE AS new\_tab –** для INSERT OR UPDATE

 **OLD TABLE AS old\_tab –** для UPDATE OR DELETE

ON mytable

**FOR EACH ROW**

EXECUTE FUNCTION fun\_foo3();

Триггерная функция.

CREATE FUNCTION fun\_foo3() RETURNS trigger

LANGUAGE plpgsql

AS $$

DECLARE

…

BEGIN

…

 **RETURN NULL**;

END;

$$;

Триггерная функция с помощью TG переменных может определить контекст вызова:

OLD – исходная строка до операции (не определена для вставки),

NEW – измененная строка (не определена при удалении),

TG\_WHEN = ‘AFTER’,

TG\_LEVEL = ‘ROW’,

TG\_OP = ‘INSERT’, ‘UPDATE’ или ‘DELETE’,

…

Доступны таблицы, содержащие не модифицированные и модифицированные строки: old\_tab и new\_tab.

### 3.2.5. Операторный триггер AFTER

Операторный триггер срабатывает после отработки всех построчных триггеров AFTER. Этот триггер вызывается один раз, независимо от количества затронутых строк. Возвращаемое значение функции игнорируется.

Операторный триггер AFTER может получить доступ к *переходным таблицам* аналогично построчному триггеру AFTER.

Триггер задается с помощью ключевых слов AFTER и FOR EACH STATEMENT.

CREATE TRIGGER tr\_foo4

**AFTER** INSERT OR UPDATE OR DELETE

**REFERENCING**

 **NEW TABLE AS new\_tab –** для INSERT OR UPDATE

 **OLD TABLE AS old\_tab –** для UPDATE OR DELETE

ON mytable

**FOR EACH STATEMENT**

EXECUTE FUNCTION fun\_foo4();

Триггерная функция.

CREATE FUNCTION fun\_foo4() RETURNS trigger

LANGUAGE plpgsql

AS $$

DECLARE

…

BEGIN

…

 **RETURN NULL**;

END;

$$;

Возвращаемое значение триггерной функции должно быть NULL. Триггерная функция с помощью TG переменных может определить контекст вызова:

TG\_WHEN = ‘BEFORE’,

TG\_LEVEL = ‘STATEMENT’,

TG\_OP = ‘INSERT’, ‘UPDATE’ или ‘DELETE’,

…

Доступны таблицы, содержащие не модифицированные и модифицированные строки: old\_tab и new\_tab.

## 3.3. Примеры триггеров

### 3.3.1. Журнал изменения строк

В БД «Авиаперевозки» с помощью триггеров будем отслеживать все изменения таблицы aircrafts (см. [3]). Создадим журнал (таблицу) aircrafts\_log.

CREATE TABLE aircrafts\_log

(LIKE aircrafts);

Добавим поля:

* date\_op –даты (и времени) выполнения операции;
* user\_op – пользователь, выполнивший операцию;
* op – операция.

 А также не уникальный индекс по столбцу aircraft\_code (хотя для примера он не обязателен, но для таблиц, подверженным частым изменениям, для ускорения выборки использование индексов будет необходимо).

ALTER TABLE aircrafts\_log

ADD COLUMN date\_op timestamp,

ADD COLUMN user\_op text,

ADD COLUMN op text;

CREATE INDEX ix\_aircraft\_code

ON aircrafts\_log(aircraft\_code);

 Записывать в журнал надо те изменения, которые были «сохранены» в таблице. Для этой цели лучше подходят операторные триггеры AFTER. Для них можно задать переходные таблицы, содержащие модифицированные, вставленные и удаленные строки. Надо иметь в виду, что использование переходных таблиц накладывает ограничения, а именно – триггеры должны быть прописаны для каждой операции отдельно. Триггерная функция будет одна.

CREATE OR REPLACE FUNCTION fun\_aircfafts\_log() RETURNS trigger

 LANGUAGE plpgsql

AS $$

BEGIN

 IF TG\_LEVEL != 'STATEMENT' OR TG\_WHEN != 'AFTER' THEN

 RAISE EXCEPTION 'Ошибка в установки триггера';

 RETURN NULL;

 END IF;

 IF TG\_OP = 'INSERT' OR TG\_OP = 'UPDATE' THEN

 INSERT INTO aircrafts\_log

 SELECT \*, now(), current\_user, TG\_OP FROM new\_tab;

 ELSE

 INSERT INTO aircrafts\_log

 SELECT \*, now(), current\_user, TG\_OP FROM old\_tab;

 END IF;

 RETURN NULL;

END;

$$;

Сначала выполняется проверка того, что триггер является операционный и AFTER. Если это не так, то формируется исключение и будет выполнен откат транзакции. Далее, для операции INSERT и UPDATE в таблицу журнала добавляем вставленные строки из переходной таблицы new\_tab вместе с текущей датой (now()), пользователем (current\_user) и операцией (TG\_OP). Для операции DELETE (для TRUNCATE не будем создавать триггер, т.к. ограничения целостности БД не позволят ее выполнить) делается тоже самое, только данные берем из переходной таблицы old\_tab.

Создадим триггер для INSERT.

CREATE TRIGGER tr\_ins\_aircfafts\_log

AFTER INSERT ON aircrafts

REFERENCING NEW TABLE AS new\_tab

FOR EACH STATEMENT

EXECUTE FUNCTION fun\_aircfafts\_log();

Добавим несколько строк, и посмотрим содержимое таблицы aircrafts\_log.

INSERT INTO aircrafts

VALUES

('TU1', 'Tupolev 1', 3000),

('TU2', 'Tupolev 2', 4000),

('TU3', 'Tupolev 3', 5000);

SELECT \* FROM aircrafts\_log;

Результат.



Триггер для UPDATE.

CREATE TRIGGER tr\_upd\_aircfafts\_log

AFTER UPDATE ON aircrafts

REFERENCING NEW TABLE AS new\_tab

FOR EACH STATEMENT

EXECUTE FUNCTION fun\_aircfafts\_log();

Проверим.

UPDATE aircrafts

 SET range = range+500,

 model = model || 'a'

WHERE aircraft\_code = 'TU2';

SELECT \* FROM aircrafts\_log;

Результат.



Теперь создадим триггер для DELETE.

CREATE TRIGGER tr\_del\_aircfafts\_log

AFTER DELETE ON aircrafts

REFERENCING OLD TABLE AS old\_tab

FOR EACH STATEMENT

EXECUTE FUNCTION fun\_aircfafts\_log();

Проверим.

DELETE FROM aircrafts

WHERE aircraft\_code = 'TU2';

SELECT \* FROM aircrafts\_log;

Результат.



### 3.3.2. Контроль ввода данных

Для таблицы aircrafts напишем триггер, который не даст завести самолет с дальностью полета (range) меньше, чем 500 км. Для начала определимся, какой триггер будем создавать. Во-первых, этот триггер должен быть установлен для операций INSERT и UPDATE, при чем для UPDATE надо контролировать изменение только столбца range. Во-вторых, этот триггер должен быть построчным. В-третьих, сделаем его – BEFORE, что позволяет отказаться о внесении изменения, не затрагивая другие строки таблицы. И в – четвертых, используем аргумент триггерной функции для передачи нижней границы дальности. Начнем с триггерной функции.

CREATE OR REPLACE FUNCTION fun\_aircfafts\_range() RETURNS trigger

 LANGUAGE plpgsql

AS $$

DECLARE range int = to\_number(TG\_ARGV[0], '99999');

BEGIN

 IF NEW.range < range THEN

 RAISE WARNING 'Для % дальность полета % меньше %',
 NEW.aircraft\_code, NEW.range, range;

 RETURN NULL;

 END IF;

 RETURN NEW;

END;

$$;

Напомним, что аргументы триггерной функции, заданные в триггере (см. ниже), передаются через массив TG\_ARGV[], отсчет индекса начинается 0. Все аргументы являются строками. Получив аргумент, преобразуем в целое число range и, далее сравниваем с введенным значением NEW.range. Если дальность меньше требуемой, формируем сообщение и отменяем изменения. Если условие по дальности выполнено, то подтверждаем изменения, возвращая NEW.

Перейдем к созданию триггера.

CREATE TRIGGER tr\_aircfafts\_range

BEFORE INSERT OR UPDATE OF **range**

ON aircrafts

FOR ROW

EXECUTE FUNCTION fun\_aircfafts\_range( **'500'**);

Обратите внимание, что для операции UPDATE задан столбец range. Тем самым, мы сообщаем системе, что триггер при UPDATE должен вызываться только тогда, когда изменяется поле range. И наконец, в триггерной функции задан параметр fun\_aircfafts\_range('500'), который заносится в элемент массива TG\_ARGV[0].

Проверим.

INSERT INTO aircrafts

 VALUES ('AN2', 'Antonov2', 400),

 ('AN3', 'Antonov3', 600);

Результат.

ПРЕДУПРЕЖДЕНИЕ: Для AN2 дальность полета 400 меньше 500

INSERT 0 1

SELECT \* FROM aircrafts



 Как видим, AN2 не попал в таблицу, выдано сообщение об ошибке, а другая запись сохранилась. Триггеры из предыдущего примера не были удалены, поэтому в журнале появилась запись для AN3.

 Проверим UPDATE.

UPDATE aircrafts

 SET range = 300

WHERE aircraft\_code = 'TU1';

Результат.

ПРЕДУПРЕЖДЕНИЕ: Для TU1 дальность полета 300 меньше 500

UPDATE 0

Триггер отработал, дальность полета для 'TU1' не изменилась.

### 3.3.3. Триггер для представления

Напишем триггер для представления. Создадим упрощенную БД «Библиотека», состоящую из трех таблиц и представления.

Читатель (reader).

CREATE TABLE reader

(

 id\_reader serial PRIMARY KEY,

 name text UNIQUE NOT NULL

);

Предполагается, что полное имя читателя (name) уникально.

Книга (book).

CREATE TABLE book

(

 id\_book serial PRIMARY KEY,

 author text NOT NULL,

 title text NOT NULL,

 CONSTRAINT ix\_author\_title UNIQUE (author, title)

);

Выданная книга (issuance).

CREATE TABLE issuance

(

 id\_issuance serial PRIMARY KEY,

 id\_reader integer NOT NULL REFERENCES reader (id\_reader),

 id\_book integer NOT NULL REFERENCES book (id\_book),

 date\_issue date NOT NULL, -- дата выдачи

 date\_return date -- дата возврата

);

Представление – книги на руках у читателя.

CREATE VIEW v\_issue\_on\_hand AS

SELECT r.id\_reader, r.name, b.author, b.title, i.date\_issue

FROM reader AS r

 LEFT JOIN issuance AS i USING (id\_reader)

 LEFT JOIN book AS b USING (id\_book)

WHERE i.date\_return IS NULL

ORDER BY r.name;

 Создадим триггер INSTEAD OF для операции INSERT для представления v\_issue\_on\_hand. Мы хотим, чтобы добавление записи отработало, как для таблицы. Например.

INSERT INTO v\_issue\_on\_hand (name, author, title, date\_issue)

 VALUES('Сидоров С.С.', 'Пушкин А.С.', 'Сказки', NOW());

Будем придерживаться следующего алгоритма, а именно,

* если читателя в таблице name нет, то его добавляем;
* если книги в таблице book нет, то формируем исключение.

Триггерная функция.

CREATE OR REPLACE FUNCTION fun\_ins\_issue\_on\_hand() RETURNS trigger

 LANGUAGE plpgsql

AS $$

DECLARE

 id\_book int;

BEGIN

 -- Ищем читателя (name – уникальное поле). Если строка
 -- найдена, то сохраним id\_ reader в NEW.id\_reader.

 SELECT id\_reader INTO NEW.id\_reader

 FROM reader

 WHERE name = NEW.name;

 IF NOT FOUND THEN

 -- Читателя нет, добавим. Для получения значения id\_reader
 -- используем предложение RETURNING INTO.
 INSERT INTO reader(name) VALUES(NEW.name)

 RETURNING id\_reader INTO NEW.id\_reader;

 END IF;

 BEGIN

 -- Поиск книги, похож на поиск читателя, кроме одного.
 -- Добавлено ключевое слово STRICT.В следствие чего,
 -- формируется исключение, если количество выбранных
 -- строк не равно 1.
 SELECT b.id\_book INTO STRICT id\_book

 FROM book b

 WHERE b.author = NEW.author AND b.title = NEW.title;

 EXCEPTION

 WHEN no\_data\_found THEN

 RAISE EXCEPTION 'Книга %, автор % отсутствует',
 NEW.title, NEW.author;

 END;

 INSERT INTO issuance(id\_reader, id\_book, date\_issue)

 VALUES( NEW.id\_reader, id\_book, NEW.date\_issue );

 RETURN NEW;

END;

$$;

 Триггер.

CREATE TRIGGER tr\_ins\_issue\_on\_hand

INSTEAD OF INSERT ON v\_issue\_on\_hand

FOR EACH ROW

EXECUTE FUNCTION fun\_ins\_issue\_on\_hand();

В заключение необходимо отметить, что триггер написан только для иллюстрации возможности применения, а не для следования этому примеру.

# 4. Роли и привилегии

## 4.1. Роли

Для того, чтобы упростить процесс управления доступом, многие СУБД предоставляют возможность объединять пользователей в группы или определять роли. Роль – это пользователь. Роль может выступать, как группа пользователей, а также включать другую групповую роль. Роли обладают атрибутами, определяющими ее особенности и права (привилегии). Например, атрибут LOGIN дает право на подключение к БД, а CREATEDB – на создание БД. Такой подход позволяет предоставить конкретному пользователю определённый набор прав в соответствии с задачами, которые на него возложены. Кроме привилегий на доступ к объектам СУБД ещё может поддерживать так называемые системные привилегии: это - права пользователя на создание/изменение/удаление (create/alter/drop) объектов различных типов.

При инсталляции СУБД PostgreSQL определяется единственная роль postgres, имеющая все права (SUPERUSER).

Для того, чтобы проверить роль, подключённого к БД пользователя, существуют две функции current\_user (или current\_role) и session\_user. Функция current\_user (роль или же текущий пользователь) возвращает имя пользователя. Функция session\_user (пользователь сеанса БД) возвращает имя пользователя, установившего текущее соединение с БД.

Любая роль может включать в себя другие роли, т. е. выступать группой. Возможно, в том числе, включение групповых ролей в другие групповые роли (но циклы не допускаются). Вообще, PostgreSQL не делает никакого различия между «обычными» и «групповыми» ролями. Смысл включения состоит в том, что для роли становятся доступны привилегии, которыми обладает групповая роль. Возможно, удобнее думать о групповой роли как о заранее сформированном наборе привилегий, который можно «выдать» обычной роли точно так же, как выдается одиночная привилегия. Это упрощает администрирование и управление доступом.

Включение в группу.

GRANT *групповая\_роль* TO *роль*;

Исключение из группы.

REVOKE *групповая\_роль* FROM *роль*;

Существует псевдороль public, которая неявно включает в себя все остальные роли. Это означает, что все роли по умолчанию будут иметь привилегии, наследуемые от public. Если выдать какие-либо привилегии роли public, эти привилегии получат вообще все роли. Правом на включение и исключение других ролей в public имеют:

* роль, включенная в public;
* роль с правами SUPERUSER;
* роль с правами CREATEROLE.

Любая роль может рассматриваться не только как пользователь СУБД,

но и может включать в себя другие роли, т. е. выступать группой. Роль

может быть включена в другую роль подобно тому, как пользователь

Unix может быть включен в группу.

Возможно в том числе включение групповых ролей в другие групповые

роли (но циклы не допускаются). Вообще, PostgreSQL не делает

никакого различия между «обычными» и «групповыми» ролями.

Смысл включения состоит в том, что для роли становятся доступны

привилегии, которыми обладает групповая роль.

Возможно, удобнее думать о групповой роли как о заранее

сформированном наборе привилегий, который можно «выдать»

обычной роли точно так же, как выдается одиночная привилегия. Это

Создадим роль pilot. Мы можем это сделать, так как роль postgres является SUPERUSER.

CREATE ROLE pilot LOGIN PASSWORD ‘12345’;

Для того, чтобы подключиться к БД, роль должна иметь атрибут LOGIN и пароль (для удаления роли используйте команду DROP ROLE *роль*). Проверим список ролей, командой \dn.



Подключимся к БД demo с ролью pilot, используя командой \connect или \c.



Проверим текущих пользователей.



## 4.2. Привилегии.

Права доступа к объектам БД определяются привилегиями. Список привилегий зависит от типа объекта БД.

С точки зрения управления доступом роли можно разбить на несколько групп.

* Суперпользователи (SUPERUSER) – полный доступ ко всем объектам, проверки не выполняются.
* Владельцы – владельцем становится тот, кто создал объект. Но право владения можно передать. Владелец имеет все привилегии на принадлежащий ему объект.
* Остальные роли – доступ только в рамках выданных привилегий на определённый объект. Такие привилегии могут выдать владельцы на свои объекты. Или может выдать суперпользователь на любой другой объект.

Чтобы проверить привилегии для pilot выполним команду \dp.



Хотя, мы дали роли pilot право на подключение к БД demo, он видит только схему public, которая доступна всем по умолчанию. Как нетрудно догадаться, надо дать привилегию на использование схемы bookings. Роль pilot сам себе, естественно, не может дать эту привилегия. Подключимся к БД с правами SUPERUSER, в данном случае, с ролью postgres и выполним команду

GRANT USAGE ON SCHEMA bookings TO pilot;

Еще раз выполним команду \dp.



В столбцах привилегий ничего нет, поэтому над этими объектами базы данных (таблицами и последовательностями) не допускаются никакие действия, если они не выполняются суперпользователем или владельцем объекта базы данных.

Посмотрим, кто имеет права на таблицу с помощью \dt.



Как видим, postgres является владельцем всех таблиц. Но, тем не менее, попробуем сделать выборку из таблицы aircrafts с привилегиями pilot.



Нет доступа.

Для предоставления привилегий на объект надо использовать команду GRANT.

GRANT *привилегии* ON *объект* TO *роль*;

Для отзыва – REVOKE.

REVOKE *привилегии* ON *объект* FROM *роль*;

 Дадим роли pilot привилегии SELECT для таблицы aircrafts и все привилегии на таблицу pilots (эти привилегии должен дать, в нашем случае postgres).

GRANT SELECT ON bookings.aircrafts TO pilot;

Для того, чтобы дать все права на таблицу pilots, можно перечислить все привилегии, как-то SELECT, INSERT, UPDATE и т.д., но более простой способ – это использовать атрибут ALL PRIVILEGES.

GRANT ALL PRIVILEGES ON bookings.pilots TO pilot;

После выполнения команд посмотрим привилегии на таблицу aircrafts, запустив команду \dp bookings.aircrafts.



И pilots.



Привилегии отображаются в формате: роль=привилегии/кем\_предоставлены.

Каждая привилегия кодируется одним символом:

* a = insert
* r = select
* w = update
* d = delete
* D = truncate
* x = reference
* t = trigger

Роль pilot теперь может с таблицей pilots выполнять любые действия, не нарушающие целостность БД, а с таблицей aircrafts только чтение – SELECT.

# 5. Практическое задание ЛР

Написать и протестировать триггеры, выполняющие следующие действия для своей предметной области. Задания формулируются на примере БД «Библиотека» вариант №1.

* Контроль соответствия дат выдачи и возврата книги.
* Контроль дублирования читателя.
* Запрет на удаления читателя, если на руках у читателя имеется книга.
* Создать таблицу, состоящую из двух целочисленных полей и содержащую одну запись, для хранения количества книг и читателей. Написать триггеры для таблиц *книги* и *читатели,* подсчитывающие при добавлении и удалении общее количество книг и читателей и, сохраняющие итоги в созданной таблице.
* Создать пользователя test и выдать ему доступ к базе данных.
* Составить и выполнить скрипты присвоения новому пользователю прав доступа к таблицам, созданным в практическом задании 1. При этом права доступа к различным таблицам должны быть различными, а именно:
	+ По крайней мере, для одной таблицы новому пользователю присваиваются права SELECT, INSERT, UPDATE в полном объеме.
	+ По крайней мере, для одной таблицы новому пользователю присваиваются права SELECT и UPDATE только избранных столбцов.
	+ По крайней мере, для одной таблицы новому пользователю присваивается только право SELECT.

# 6. Темы для самостоятельной проработки

* Массивы
* https://postgrespro.ru/docs/postgrespro/11/arrays https://postgrespro.ru/docs/postgrespro/11/functions-array
* Процедурный язык PL/pgSQL.
* [https://postgrespro.ru/docs/postgrespro/11/plpgsql](https://postgrespro.ru/docs/postgrespro/11/plpgsql%20)
* Курсоры.
* <https://postgrespro.ru/docs/postgrespro/11/plpgsql-cursors>
* Триггеры.
* https://postgrespro.ru/docs/postgrespro/11/triggers
* https://postgrespro.ru/docs/postgrespro/11/plpgsql-trigger
* Роли и пользователи.
* https://postgrespro.ru/docs/postgresql/11/user-manag https://postgrespro.ru/docs/postgrespro/11/app-createuser
* Директивы GRANT и REVOKE.
* https://postgrespro.ru/docs/postgrespro/11/ddl-priv https://postgrespro.ru/docs/postgrespro/11/ddl-schemas#DDL-SCHEMAS-PRIV

# 6. Примеры вопросов для самостоятельной проработки

* В каких случаях целесообразно использовать массивы в таблицах БД?
* Рассказать о курсорах, как и зачем используются.
* Для каких операций над таблицами можно создать триггеры?
* Для каких событий можно создать триггеры?
* Сколько триггеров можно создать для таблицы?
* Что возвращает триггерная функция?
* В каком триггере и как отменить операцию для строки без аварийного завершения операции?
* Для чего нужно условие WHEN и где целесообразно его использовать?
* Как создать триггер, который вызывается только при изменении конкретных столбцов таблицы?
* Как передаются параметры триггерной функции?
* Какие специальные переменные TG\_ … доступны триггерной функции?
* Для каких событий доступны переходные таблицы и что они содержат?
* Для чего нужны роли?
* Что такое схема?
* Рассказать про директивы GRANT и REVOKE.
* Для чего нужна роль PUBLIC?
* Как добавить нового пользователя в текущую базу данных?
* Как позволить пользователю заходить на сервер?
* Какие существуют права?
* Исправить ошибки в обязательной части.
* Сменить владельца базы данных.
* Сменить пароль для пользователя.
* Определить роль с заданными правами.

# ПИСОК 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.