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 №4 «Массивы, курсоры, триггеры, роли»

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 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 использоваться 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.

Поэлементная 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 pilots  VALUES  (‘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 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 номером abs(*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;

$$;

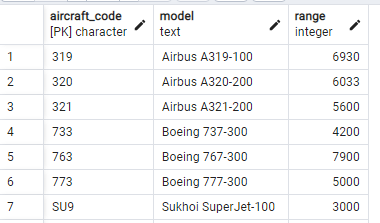
Вызовем.

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;

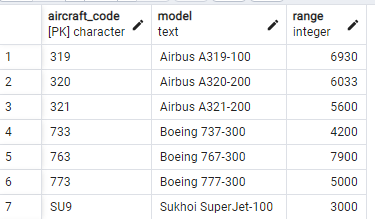
$$;

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 1 перечисляются типы триггеров.

**Таблице 1. 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

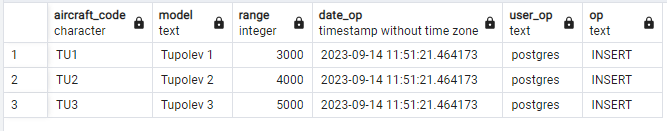
('TU1', 'Tupolev 1', 3000),

('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

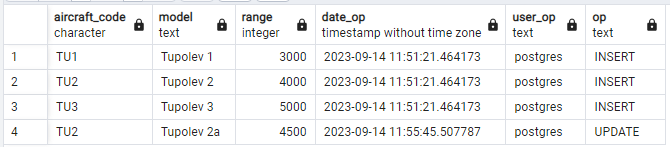
SET range = range+500,

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();

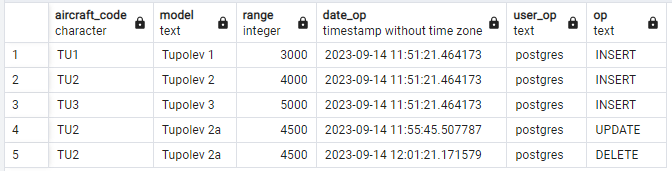
Проверим.

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),

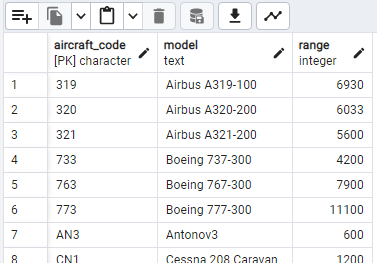
('AN3', 'Antonov3', 600);

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

Одним из основных способов обеспечения конфиденциальности и целостности информации в БД выступает механизм аутентификации. Аутентификация– это процедура проверки подлинности пользователя (точнее, его идентификатора). Обычно пользователь подтверждает то, что он является именно тем, за кого он себя выдает, путем ввода в систему уникальной (неизвестной другим) информации о себе. В простейшем случае это символьный пароль, но возможен и более сложный подход подтверждения подлинности, в том числе биометрическая аутентификация или электронные способы аутентификации. Если пользователь успешно прошел процедуру аутентификации, СУБД осуществляет его авторизацию. Авторизация– это процедура предоставления пользователю определенных ресурсов и прав на их использование. Все дальнейшее взаимодействие пользователя с объектами БД строго регламентируется в соответствии с назначенными правами. Пользователи могут обладать разными правами на один и тот же объект, одни могут лишь просматривать данные, другие – только добавлять, третьи – осуществлять чтение, вставку и редактирование.

Привилегиями доступаназывают разрешение на использование определенной услуги управления данными для доступа к объекту данных (например, чтение, запись или исполнение), предоставляемое идентифицированному пользователю.

PostgreSQL управляет правами доступа к базе данных, используя концепцию ролей. Роль может рассматриваться как пользователь базы данных или как группа пользователей базы данных, в зависимости от настройки роли. Роли могут владеть объектами базы данных (например, таблицами и функциями) и назначать привилегии на эти объекты другим ролям, чтобы контролировать доступ к тем или иным объектам. Кроме того, можно предоставить членство в роли другой роли, что позволит роли-участнику использовать привилегии, назначенные другой роли.

Концепция ролей объединяет понятия «пользователи» и «группы». То есть нет пользователей и групп, есть только роли. Любая роль может выполнять функции пользователя, группы или и того, и другого.

Так как привилегии не назначаются пользователям непосредственно и приобретаются ими только через свою роль (или роли), управление индивидуальными правами пользователя, по сути, сводится к назначению ему ролей. Это упрощает такие операции, как добавление пользователя или смена подразделения пользователем.

Так, например, можно определить роли «Администратор», «Аналитик», «Пользователь» и в дальнейшем управлять доступом, используя роли, а не адресные привилегии доступа к объектам.

**При создании объекта БД ему назначается владелец. Обычно владельцем является роль, выполнившая оператор создания. Для большинства типов объектов изначально только владелец (или суперпользователь) может выполнять какие-либо действия с объектом. Чтобы разрешить другим ролям использовать объект, необходимо предоставить соответствующие привилегии*.***

Существуют различные виды привилегий:  
SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, CREATE, CONNECT, TEMPORARY, EXECUTE, USAGE, SET, ALTER SYSTEM, и MAINTAIN. Привилегии, применяемые к конкретному объекту БД, различаются в зависимости от его типа (таблица, функция и т. д.).

4.1. Создание и модификация пользователей

Роли базы данных являются глобальными для всей установки кластера базы данных (не для отдельной базы данных). Для создания роли используется команда SQL CREATE ROLE:

CREATE ROLE *имя* [ [ WITH ] *параметр* [ ... ] ]

**Параметры**

SUPERUSER | NOSUPERUSER

| CREATEDB | NOCREATEDB

| CREATEROLE | NOCREATEROLE

| INHERIT | NOINHERIT

| LOGIN | NOLOGIN

| REPLICATION | NOREPLICATION

| CONNECTION LIMIT предел\_подключений

| PASSWORD 'пароль' | PASSWORD NULL

| VALID UNTIL 'дата\_время'

| IN ROLE имя\_роли [, ...]

| ROLE имя\_роли [, ...]

| ADMIN имя\_роли [, ...]

Прежде всего отметим, что бывают обычные пользователи и суперпользователи. Если задан атрибут SUPERUSER, то ограничения, налагаемые на обычного пользователя, не действуют. Суперпользователь может удалять любые объекты (базы данных и т. д.), какие захочет.

Важный момент: для создания базы данных у пользователя должны быть права на уровне экземпляра.

В команде после имени можно указать параметры создаваемой роли. В частности, можно указать параметр LOGIN, определяющий, разрешается ли новой роли вход на сервер, то есть, может ли эта роль стать начальным авторизованным именем при подключении клиента.

Существует также команда CREATE USER, которая является синонимом команды CREATE ROLE:

CREATE USER *имя* [ [ WITH ] *параметр* [ ... ] ]

…

| LOGIN | NOLOGIN

Единственное отличие между командами состоит в том, что для команды CREATE USER по умолчанию подразумевается использование параметра LOGIN, а в виде CREATE ROLE по умолчанию используется NOLOGIN.

**Параметры**

*имя -* имя создаваемой роли.

SUPERUSER, NOSUPERUSER - эти предложения определяют, будет ли эта роль «суперпользователем», который может переопределить все ограничения доступа в базе данных. Статус суперпользователя несёт опасность и назначать его следует только в случае необходимости. Создать нового суперпользователя может только суперпользователь. В отсутствие этих предложений по умолчанию подразумевается NOSUPERUSER.

CREATEDB, NOCREATEDB – эти предложения определяют, сможет ли роль создавать базы данных. Указание CREATEDB даёт новой роли это право, а NOCREATEDB запрещает роли создавать базы данных. По умолчанию подразумевается NOCREATEDB. Только суперпользователи или роли с правом CREATEDB могут указывать CREATEDB.

CREATEROLE, NOCREATEROLE - эти предложения определяют, сможет ли роль создавать, изменять и удалять другие роли, добавлять для них комментарии и изменять метку безопасности в любых ролях. Подробнее возможности, предоставляемые этим правом, описаны в Создание роли. Если предложение не указано, по умолчанию используется NOCREATEROLE.

INHERIT, NOINHERIT – эти предложения влияют, будет ли роль «наследовать» привилегии ролей, членом которых она является. Роль с атрибутом INHERIT может автоматически использовать любые права, назначенные всем ролям, в которые она включена, непосредственно или опосредованно. Иначе права другой роли можно использовать только после переключения на нее командой SET ROLE.

LOGIN, NOLOGIN – эти предложения определяют, разрешается ли новой роли вход на сервер; то есть, может ли эта роль стать начальным авторизованным именем при подключении клиента. Можно считать, что роль с атрибутом LOGIN соответствует пользователю. Роли без этого атрибута бывают полезны для управления доступом в базе данных, но это не пользователи в обычном понимании. По умолчанию подразумевается вариант NOLOGIN, за исключением вызова CREATE ROLE в виде CREATE USER.

REPLICATION, NOREPLICATION – эти предложения определяют, будет ли роль ролью репликации. Чтобы роль могла подключаться к серверу в режиме репликации (в режиме физической или логической репликации) и создавать/удалять слоты репликации, у неё должен быть этот атрибут (либо это должна быть роль суперпользователя). Роль, имеющая атрибут REPLICATION, обладает очень большими привилегиями, и, поэтому, этот атрибут должны иметь только роли, фактически используемые для репликации. По умолчанию подразумевается вариант NOREPLICATION. Выдавать атрибут REPLICATION могут только суперпользователи или роли с REPLICATION.

BYPASSRLS, NOBYPASSRLS – эти предложения определяют, будут ли для роли игнорироваться все политики защиты на уровне строк (RLS). По умолчанию подразумевается вариант NOBYPASSRLS. Выдавать атрибут BYPASSRLS могут только суперпользователи или роли с BYPASSRLS.

CONNECTION LIMIT *предел\_подключений* – если роли разрешён вход, этот параметр определяет, сколько параллельных подключений может установить роль. Значение -1 (по умолчанию) снимает ограничение. Заметьте, что под это ограничение подпадают только обычные подключения. Ни подготовленные транзакции, ни соединения фоновых рабочих процессов в расчёт не берутся.

PASSWORD *'пароль'*, PASSWORD NULL – задаёт пароль роли. (Пароль полезен только для ролей с атрибутом LOGIN, но задать его можно и для ролей без такого атрибута.) Если проверка подлинности по паролю не будет использоваться, этот параметр можно опустить. При указании пустого значения будет задан пароль NULL, что не позволит данному пользователю пройти проверку подлинности по паролю. При желании пароль NULL можно установить явно, указав PASSWORD NULL.

VALID UNTIL *'дата\_время'* – предложение VALID UNTIL устанавливает дату и время, после которого пароль роли перестаёт действовать. Если это предложение отсутствует, срок действия пароля будет неограниченным.

IN ROLE *имя\_роли***–** с предложением IN ROLE новая роль автоматически становится членом указанных существующих ролей.

ROLE *имя\_роли*–с предложением ROLE одна или нескольких указанных существующих ролей автоматически добавляются как члены новой роли.

ADMIN *имя\_роли*–предложение ADMIN работает подобно ROLE, но перечисленные в нём роли добавляются как члены новой роли с включённым атрибутом ADMIN, что даёт им право включать в новою роль другие роли.

Создадим несколько ролей. Начнем с создания групповой роли admin, в которую будем включать пользователей с привилегией SUPERUSER. Подсоединимся к любой БД сервера с правами SUPERUSER (по умолчанию это postgres), с помощью утилиты psql (рис. 4.1.1):

Пользователь postgres

Server [localhost]:

Database [postgres]:

Port [5432]:

Username [postgres]:

Пароль пользователя postgres:

psql (16.8)

ПРЕДУПРЕЖДЕНИЕ: Кодовая страница консоли (866) отличается от основной

страницы Windows (1251).

8-битовые (русские) символы могут отображаться некорректно.

Подробнее об этом смотрите документацию psql, раздел

"Notes for Windows users".

Введите "help", чтобы получить справку.

postgres=# \! chcp 1251

Текущая кодовая страница: 1251

postgres=#

Рисунок 4.1.1

Выполним команду CREATE ROLE (рис. 4.1.2):

Пользователь postgres

postgres=# CREATE ROLE admin WITH SUPERUSER;

CREATE ROLE

postgres=#

Рисунок 4.1.2

Этой командой мы создали роль суперпользователя, но без права подключения к серверу. Это обычная практика при создании групповой роли, групповая роль не должна иметь логин и пароль.

Следующим шагом, добавим пользователя adm1 в группу admin  
 (рис. 4.1.3):

Пользователь postgres

postgres=# CREATE ROLE adm1 WITH IN ROLE admin LOGIN PASSWORD '12345';

CREATE ROLE

postgres=#

Рисунок 4.1.3

Пользователь adm1 имеет право подключиться к серверу с заданным паролем.

Запустим еще одну утилиту psql от имени adm1 и создадим тестовую БД (рис. 4.1.4):

Пользователь adm1

Server [localhost]:

Database [postgres]:

Port [5432]:

Username [postgres]: adm1

Пароль пользователя adm1:

psql (16.8)

ПРЕДУПРЕЖДЕНИЕ: Кодовая страница консоли (866) отличается от основной

страницы Windows (1251).

8-битовые (русские) символы могут отображаться некорректно.

Подробнее об этом смотрите документацию psql, раздел

"Notes for Windows users".

Введите "help", чтобы получить справку.

postgres=> \! chcp 1251

Текущая кодовая страница: 1251

postgres=> CREATE DATABASE db\_test;

ОШИБКА: нет прав для создания базы данных

postgres=>

Рисунок 4.1.4

У суперпользователя нет прав! Может нам удастся создать таблицу в БД postgres (рис. 4.1.5):

Пользователь adm1

postgres=> CREATE TABLE test (id int);

ОШИБКА: нет доступа к схеме public

СТРОКА 1: CREATE TABLE test (id int);

^

postgres=>

Рисунок 4.1.5

И здесь неудача. Все дело в том, что привилегии роли LOGIN, SUPERUSER, CREATEDB и CREATEROLE никогда не наследуются как обычные права на объекты базы данных. Чтобы ими воспользоваться, необходимо переключиться на роль, имеющую этот атрибут, с помощью команды SET ROLE. Таким образом мы, подключившись, как adm1, должны переключится на роль admin и внести изменения, что и сделаем   
(рис. 4.1.6):

Пользователь adm1

postgres=> SET ROLE admin;

SET

postgres=# CREATE DATABASE db\_test;

CREATE DATABASE

postgres=# \l

Список баз данных

Имя | Владелец | Кодировка | Провайдер локали | LC\_COLLATE |

-----------+----------+-----------+------------------+------------+

db\_test | admin | UTF8 | libc | ru-RU |

demo | postgres | UTF8 | libc | ru-RU |

postgres | postgres | UTF8 | libc | ru-RU |

template0 | postgres | UTF8 | libc | ru-RU |

| | | | |

template1 | postgres | UTF8 | libc | ru-RU |

| | | | |

(5 строк)

postgres=#

Рисунок 4.1.6

Работает, но вводить лишнюю команду, имея хоть и опосредованно привилегию суперпользователя, значит себя не уважать. Воспользуемся командой:

ALTER ROLE *указание\_роли* [ WITH ] *параметр* [ ... ]

**Параметры**

SUPERUSER | NOSUPERUSER

| CREATEDB | NOCREATEDB

| CREATEROLE | NOCREATEROLE

| INHERIT | NOINHERIT

| LOGIN | NOLOGIN

| REPLICATION | NOREPLICATION

| BYPASSRLS | NOBYPASSRLS

| CONNECTION LIMIT *предел\_подключений*

| PASSWORD '*пароль*' | PASSWORD NULL

| VALID UNTIL '*дата\_время*'

Перечисленные параметры имеют те же значение что и при создании роли.

Мы подсоединились к серверу (рис. 4.1.6) и переключились на роль admin, которая, является суперпользователем и тем самым имеет возможность назначить привилегию SUPERUSER роли adm1 (рис. 4.1.7):

postgres=# ALTER ROLE adm1 WITH SUPERUSER;

ALTER ROLE

Рисунок 4.1.7

Переключимся обратно на роль adm1 и создадим БД (рис. 4.1.8):

postgres=# SET ROLE adm1;

SET

postgres=# CREATE DATABASE db\_test3;

CREATE DATABASE

Рисунок 4.1.8

Утилита psql имеет команду \du для просмотра ролей сервера, введем ее (рис. 4.1.9):

postgres=# \du

Список ролей

Имя роли | Атрибуты

----------+------------------------------------------------------------------------

adm1 | Суперпользователь

admin | Суперпользователь, Вход запрещён

postgres | Суперпользователь, Создаёт роли, Создаёт БД, Репликация, Пропускать RLS

postgres=#

Рисунок 4.1.9

После получения прав суперпользователя, желательно изменить пароль, если он был вам выдан. Пароль меняется так же с помощью ALTER ROLE:

ALTER ROLE adm1 WITH PASSWORD 'новый пароль';

Создадим себе помощника adm2, включим его в группу admin и сделаем его суперпользователем. Для добавления и удаления роли из группы используется команда ALTER GROUP:

ALTER GROUP *указание\_роли* ADD USER *имя\_пользователя* [, ... ]

ALTER GROUP *указание\_роли* DROP USER *имя\_пользователя* [, ... ]

Здесь *указание\_роли*:

имя\_роли

| CURRENT\_ROLE

| CURRENT\_USER

| SESSION\_USER

Выполним задуманное (рис. 4.1.10):

Пользователь adm1

postgres=# CREATE ROLE adm2 WITH SUPERUSER LOGIN PASSWORD '23456';

CREATE ROLE

postgres=# ALTER GROUP admin ADD USER adm2;

ALTER ROLE

postgres=# \du

Список ролей

Имя роли | Атрибуты

----------+------------------------------------------------------------------------

adm1 | Суперпользователь

adm2 | Суперпользователь

admin | Суперпользователь, Вход запрещён

postgres | Суперпользователь, Создаёт роли, Создаёт БД, Репликация, Пропускать RLS

postgres=#

Рисунок 4.1.10

Заведем еще одного пользователя adm3 с привилегией SUPERUSER и добавим в роль admin, но с помощью pgAdmin 4[[1]](#footnote-1). Запустим pgAdmin от имени суперпользователя. Развернем дерево объектов в левой панели (рис. 4.1.11):

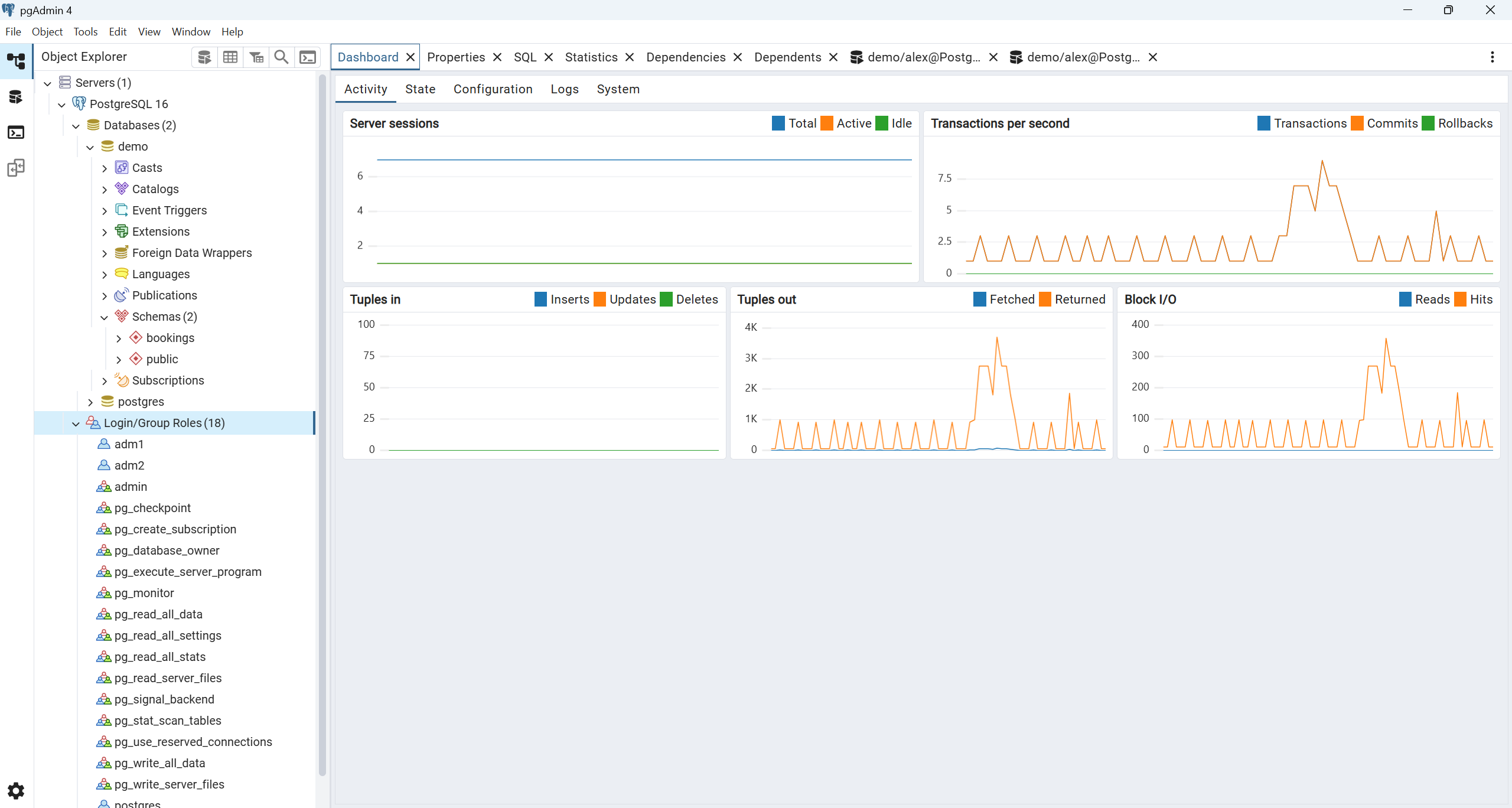


Рисунок 1. 1.11

Щелкнем ПКМ[[2]](#footnote-2) по узлу *Login/Group Roles* и выполним команду   
*Create-> Login/Group Role…* (рис. 4.1.12):

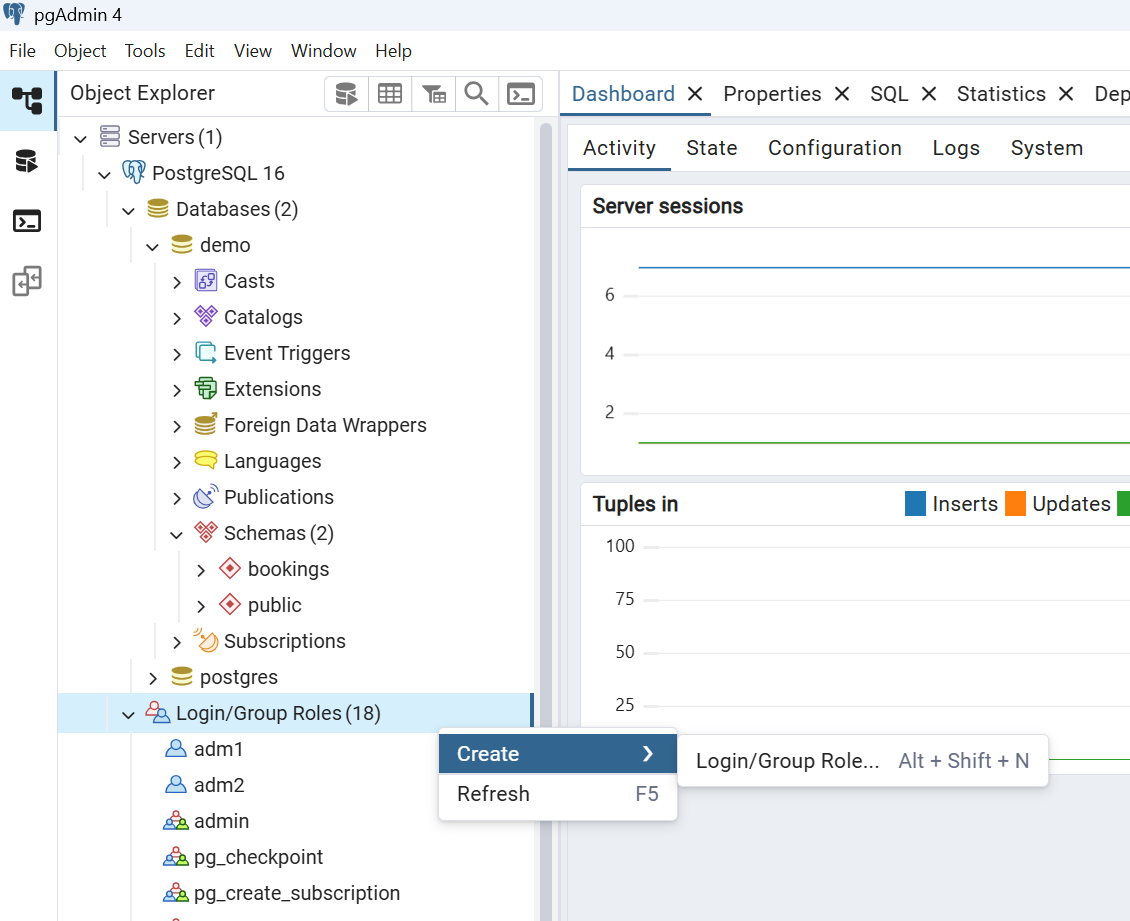


Рисунок 4.1.12

Откроется диалог *Group Role – Login/Group Roles* (рис. 4.1.13):

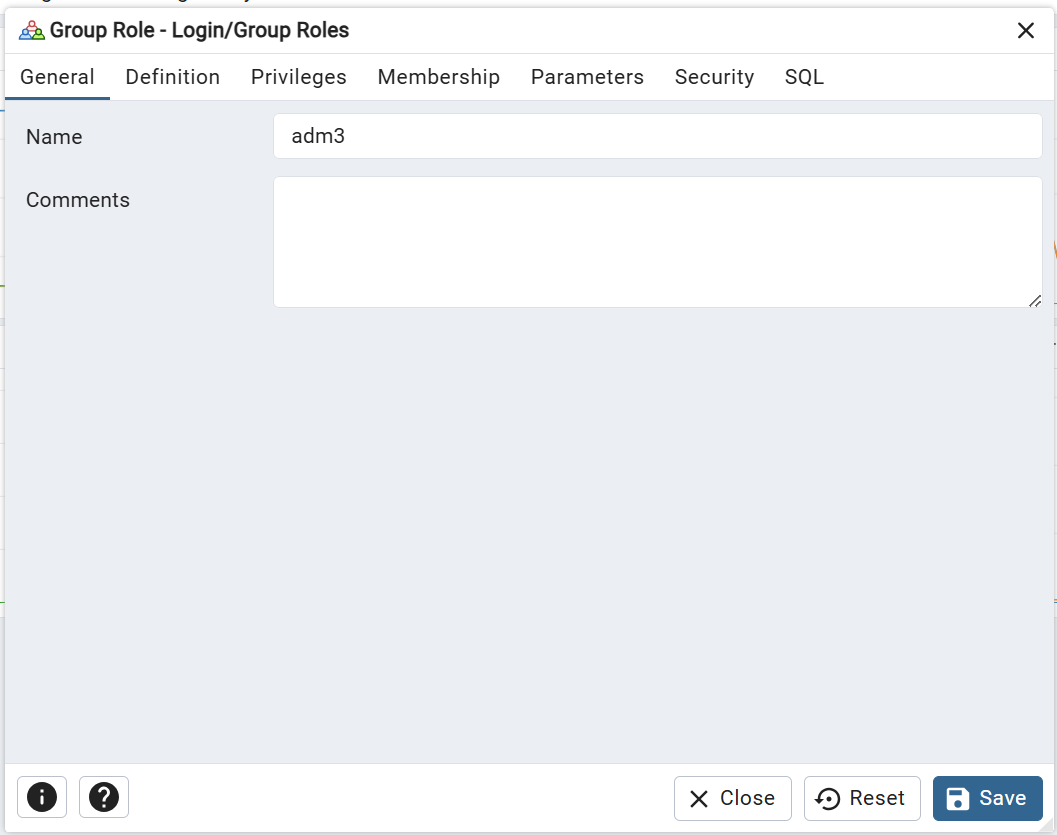


Рисунок 4.1.13

В поле Name имя пользователя и перейдем на вкладку *Definition*   
(рис. 4.1.14):

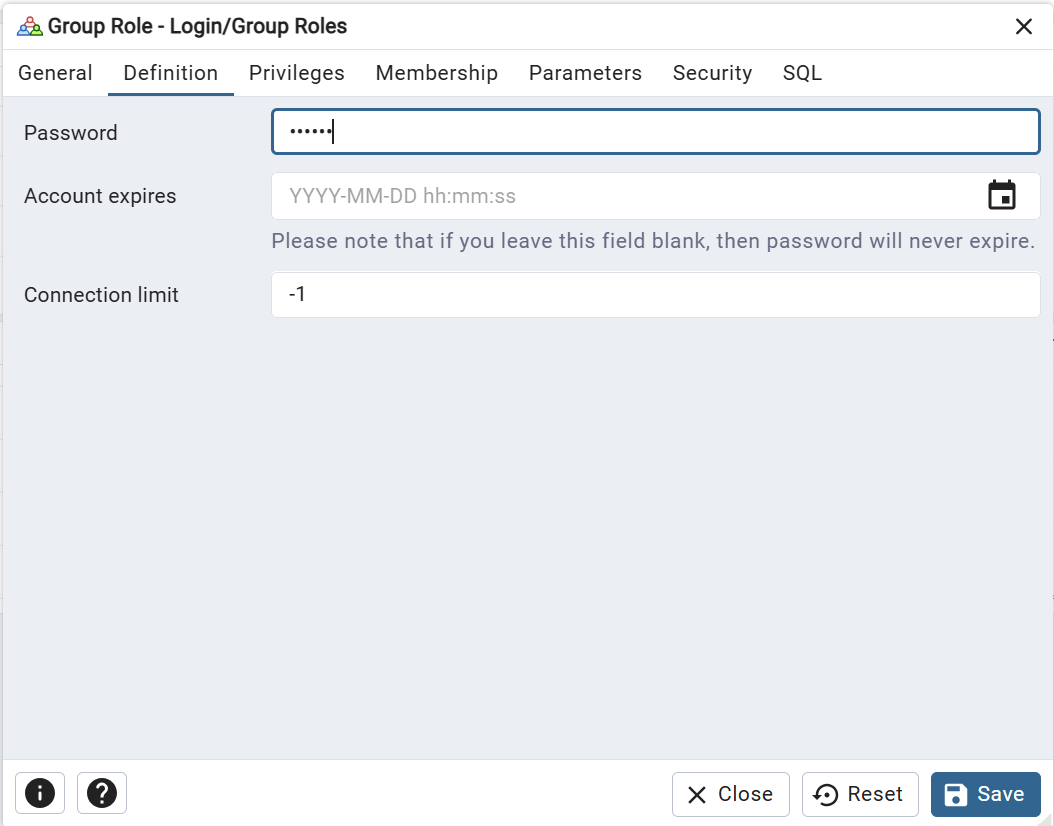


Рисунок 4.1.14

Зададим пароль и далее вкладка *Privileges* (рис. 4.1.15):

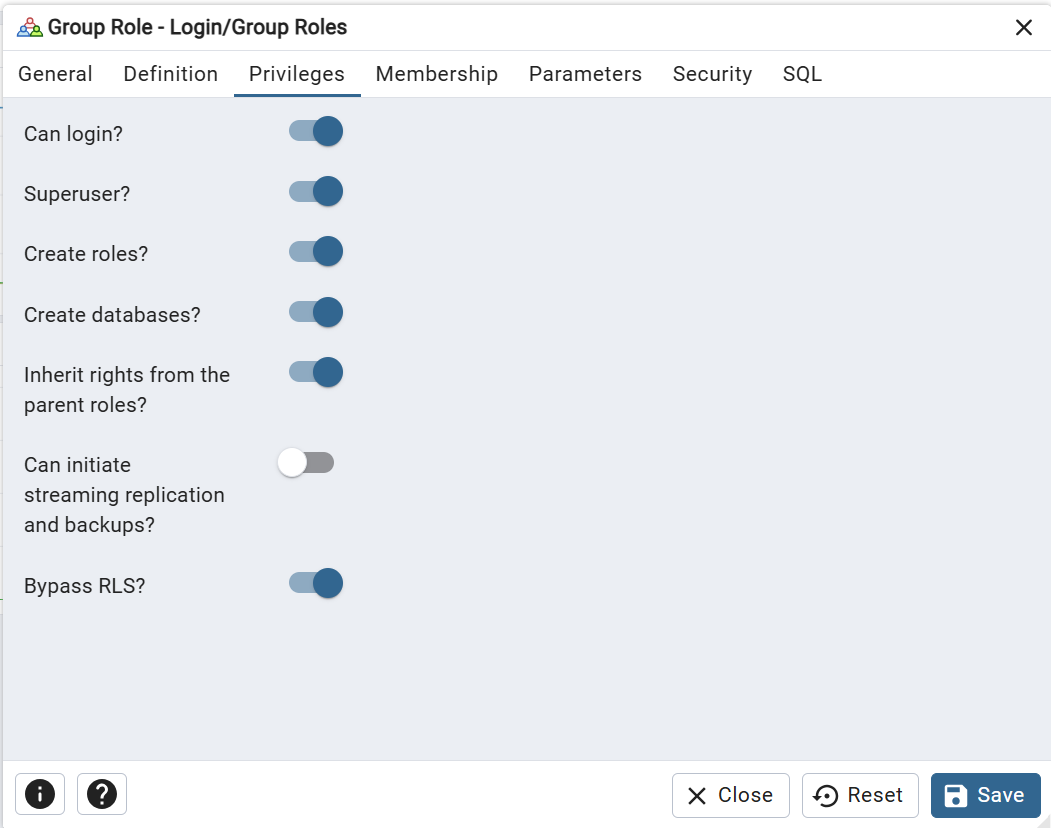


Рисунок 4.1.15

Разрешим adm3 все, кроме репликации и backups. На следующей вкладке *Membership* сделаем его членом группы admin. Роль, членом которой хотим сделать adm3 выбирается из выпадающего списка, в котором наряду с предопределёнными групповыми ролями, перечислены и роли, созданные нами (рис. 4.1.16):

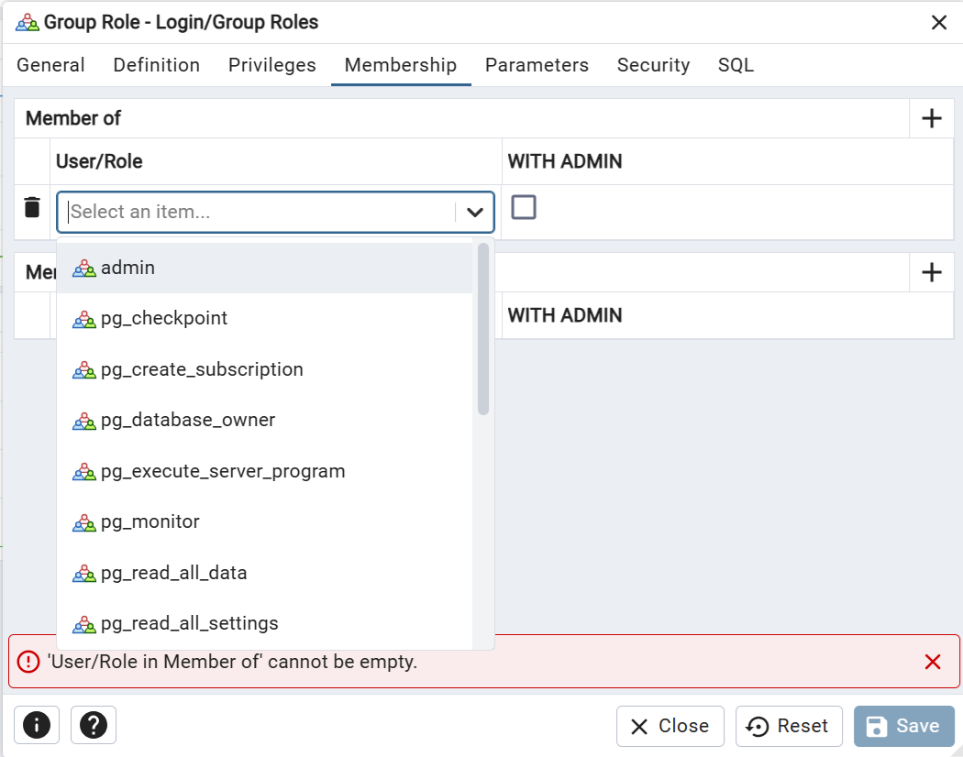


Рисунок 4.1.16

Следующие две вкладки мы пропустим и перейдем к вкладке *SQL* (рис. 4.1.17):

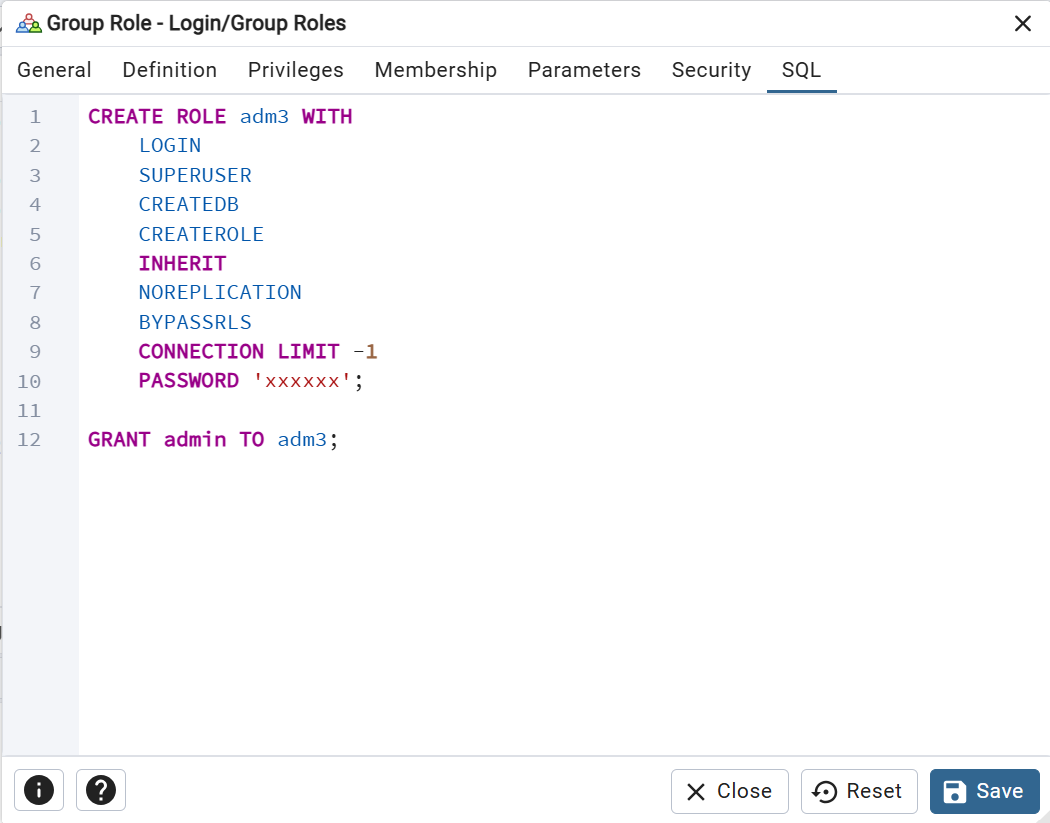


Рисунок 4.1.17

В скрипте перечислены все привилегии, кроме репликации и предложение, включающее adm3 в группу admin.

Зайдем в свойства (Properties) групповой роли admin (щелкнув по ней ПКМ) и посмотрим ее состав (рис. 4.1.18):

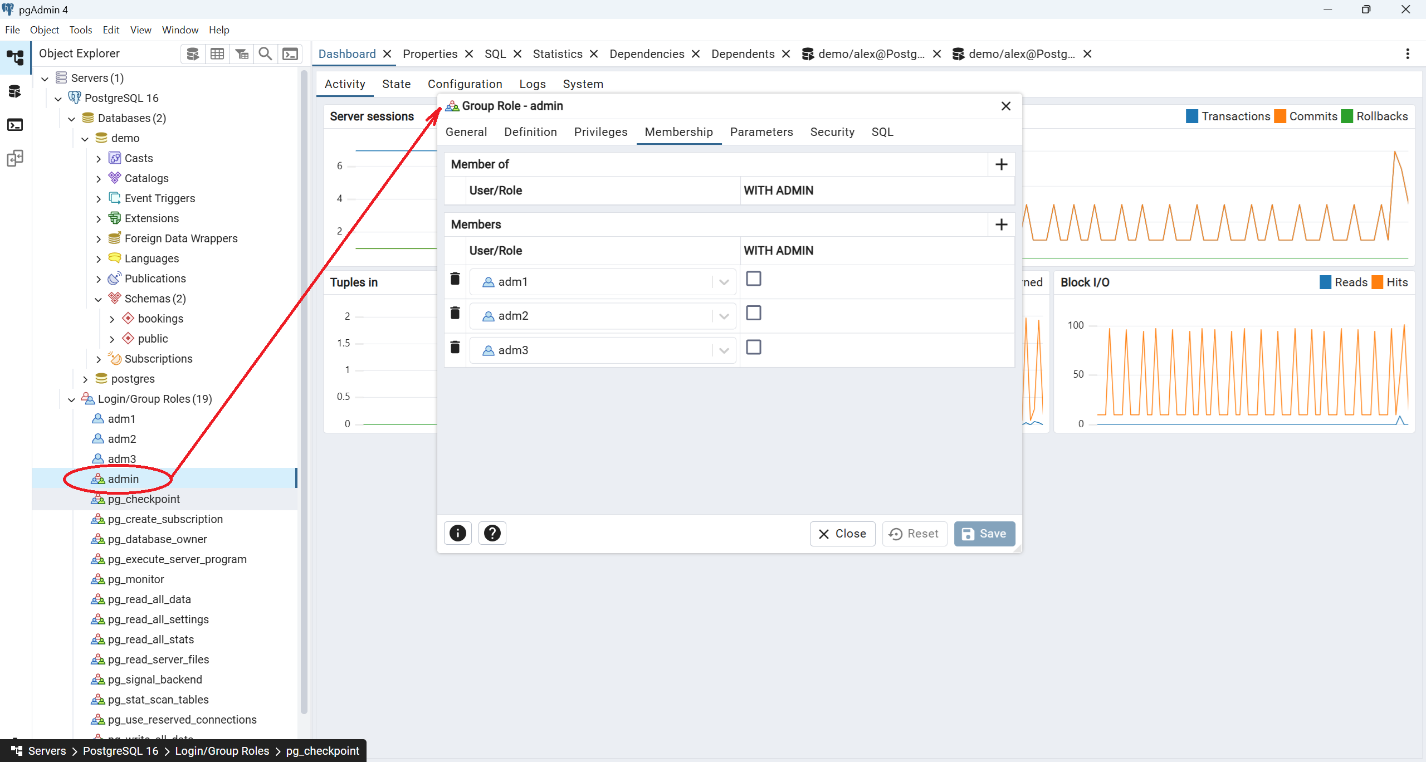


Рисунок 4.1.18

Результат ожидаемый.

4.2. Безопасность на уровне базы данных

Заведем пользователей, которые могут подсоединиться к БД demo и читать данные таблиц, т. е. выполнять SELECT. Начнем с подключения к БД. Для баз данных команда GRANT позволяет задать следующие права доступа:

GRANT {{ CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [PRIVILEGES]}

ON DATABASE *имя\_бд* [, ...]

TO *указание\_роли* [, ...] [ WITH GRANT OPTION ]

**Параметры**

CREATE – это право позволяет создавать схемы в конкретной базе. Отметим, чтоCREATE не разрешает создавать таблицы.

CONNECT [–](https://postgrespro.ru/docs/postgresql/current/ddl-priv#DDL-PRIV-TEMPORARY) позволяет подключаться к базе данных. Это право проверяется при   
установлении соединения.

TEMPORARY или TEMP [–](https://postgrespro.ru/docs/postgresql/current/ddl-priv#DDL-PRIV-TEMPORARY) позволяет создавать временные таблицы в определённой базе данных.

Обратная команда:

REVOKE [ GRANT OPTION FOR ]

{{ CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ]}

ON DATABASE *имя\_бд* [, ...]

FROM *указание\_роли* [, ...]

[ CASCADE ]

**Параметры**

GRANT OPTION FOR – если указана эта опция, то отзывается только право передачи права, но не само право. Без этого указания отзывается и право, и право распоряжаться им.

CASCADE – если пользователь обладает правом с правом передачи и он дал его другим пользователям, последнее право считается зависимым. Когда первый пользователь лишается самого права или права передачи и существуют зависимые права, эти зависимые права также отзываются, если дополнительно указано CASCADE; в противном случае операция завершается ошибкой.

Приступим (рис. 4.2.1):

Пользователь postgres

demo=# CREATE ROLE readers;

CREATE ROLE

demo=# CREATE ROLE fio1 WITH IN ROLE readers LOGIN PASSWORD '123';

CREATE ROLE

demo=# \du

Список ролей

Имя роли | Атрибуты

----------+------------------------------------------------------------------------

adm1 | Суперпользователь

adm2 | Суперпользователь

adm3 | Суперпользователь, Создаёт роли, Создаёт БД, Пропускать RLS

admin | Суперпользователь, Вход запрещён

fio1 |

postgres | Суперпользователь, Создаёт роли, Создаёт БД, Репликация, Пропускать RLS  
 readers | Вход запрещён

Рисунок 4.2.1

Пользователь fio1 пока не имеет прав для подключения к БД demo, проверим. Зайдем с другого терминала и подключимся от имени пользователя fio1   
(рис. 4.2.2):

Пользователь fio1

Server [localhost]:

Database [postgres]: demo

Port [5432]:

Username [postgres]: fio1

Пароль пользователя fio1:

psql (16.8)

ПРЕДУПРЕЖДЕНИЕ: Кодовая страница консоли (866) отличается от основной

страницы Windows (1251).

8-битовые (русские) символы могут отображаться некорректно.

Подробнее об этом смотрите документацию psql, раздел

"Notes for Windows users".

Введите "help", чтобы получить справку.

demo=>

Рисунок 4.2.2

Подсоединились! В чем дело? А дело в том, что каждая БД имеет специальную групповую роль public, в которую автоматом включаются все роли и которая имеет привилегию CONNECT. Чтобы этого не происходило необходимо отозвать эту привилегию у public (рис. 4.2.3):

Пользователь postgres

demo=# REVOKE ALL ON DATABASE demo FROM public;

REVOKE

demo=#

Рисунок 4.2.3

Повторим попытку подключения (рис. 4.2.4):

Пользователь fio1

Server [localhost]:

Database [postgres]: demo

Port [5432]:

Username [postgres]: fio1

Пароль пользователя fio1:

psql: ошибка: подключиться к серверу "localhost" (::1), порту 5432 не удалось: ВАЖНО: доступ к базе "demo" запрещён

ПОДРОБНОСТИ: Пользователь не имеет привилегии CONNECT.

Для продолжения нажмите любую клавишу . . .

Рисунок 4.2.4

Предоставим право подключения роли readers к БД demo (рис. 4.2.5):

Пользователь postgres

demo=# GRANT CONNECT ON DATABASE demo TO readers;

GRANT

demo=#

Рисунок 4.2.5

Войдем с правами fio1 и проведем пару тестов (рис. 4.2.6):

Пользователь fio1

Server [localhost]:

Database [postgres]: demo

Port [5432]:

Username [postgres]: fio1

Пароль пользователя fio1:

psql (16.8)

ПРЕДУПРЕЖДЕНИЕ: Кодовая страница консоли (866) отличается от основной

страницы Windows (1251).

8-битовые (русские) символы могут отображаться некорректно.

Подробнее об этом смотрите документацию psql, раздел

"Notes for Windows users".

Введите "help", чтобы получить справку.

demo=> \! chcp 1251

Текущая кодовая страница: 1251

demo=> SELECT \* FROM bookings.aircrafts\_data;

ОШИБКА: нет доступа к схеме bookings

СТРОКА 1: SELECT \* FROM bookings.aircrafts\_data;

^

demo=> CREATE SCHEMA rrr;

ОШИБКА: нет доступа к базе данных demo

demo=> CREATE table test (id int);

ОШИБКА: нет доступа к схеме public

СТРОКА 1: CREATE table test (id int);

^

demo=>

Рисунок 4.2.6

Пользователь fio1 еще не имеет возможности получить данные – нет доступа к схеме bookings.

Перед тем, как приступить к предоставлению привилегий доступа к схеме, рассмотрим, как выполнить предыдущие операции с помощью pgAdmin.

Откроем pgAdmin с правами суперпользователя и создадим групповую роль writer\_aircrafts (без привилегии LOGIN), содержащая пользователя foi2 (рис. 4.2.7).

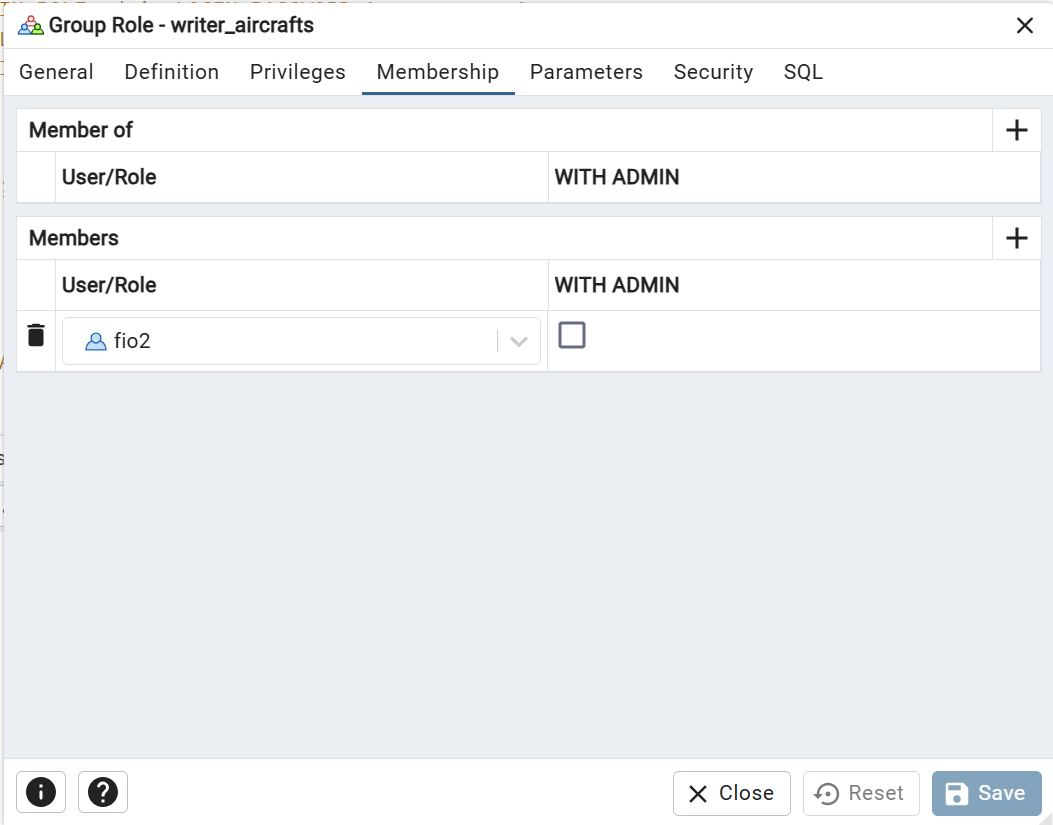


Рисунок 4.2.7

Групповую роль writer\_aircrafts наделим привилегиями, позволяющими редактировать записи таблицы aircrafts\_data. Но сначала необходимо предоставить возможность подключиться к БД demo.

Щелкнем ПКМ по узлу demo на панели объектов и выберем меню *Properties…*. Откроется диалог, в котором выберем вкладку Security   
(рис. 4.2.8):

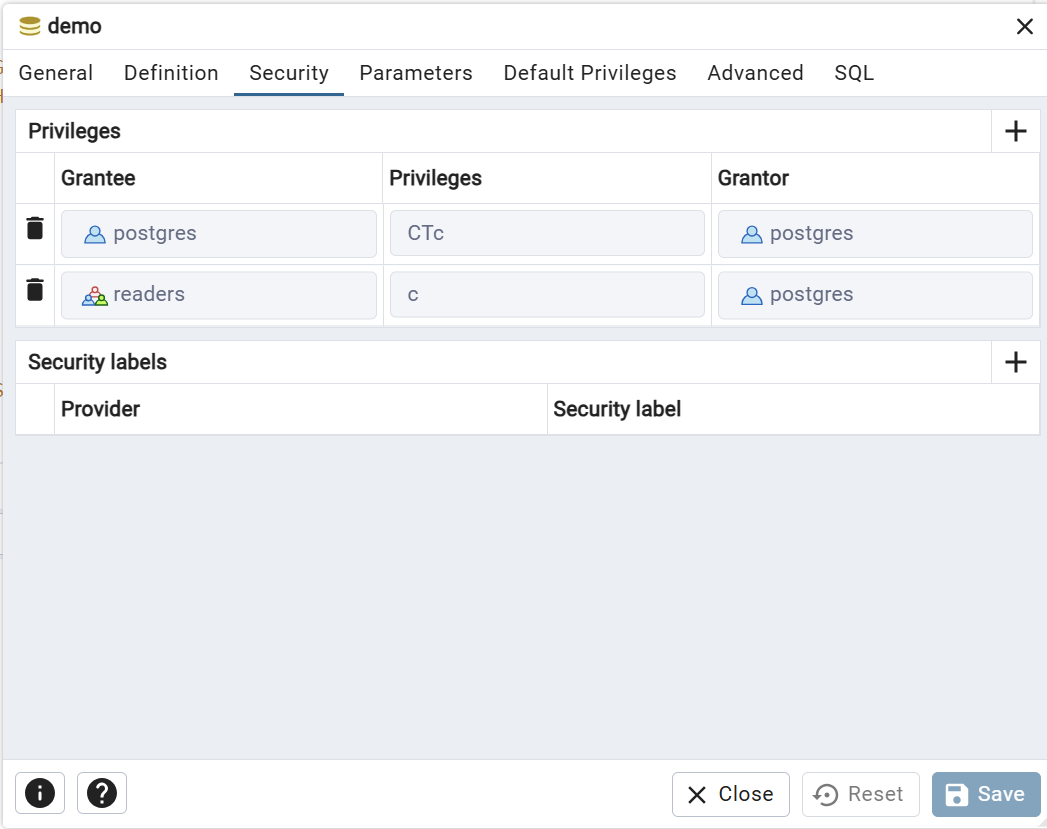


Рисунок 4.2.8

Нажмем на «+», чтобы добавить роль writer\_aircrafts и наделим ее привилегией CONNECT (рис. 4.2.9):

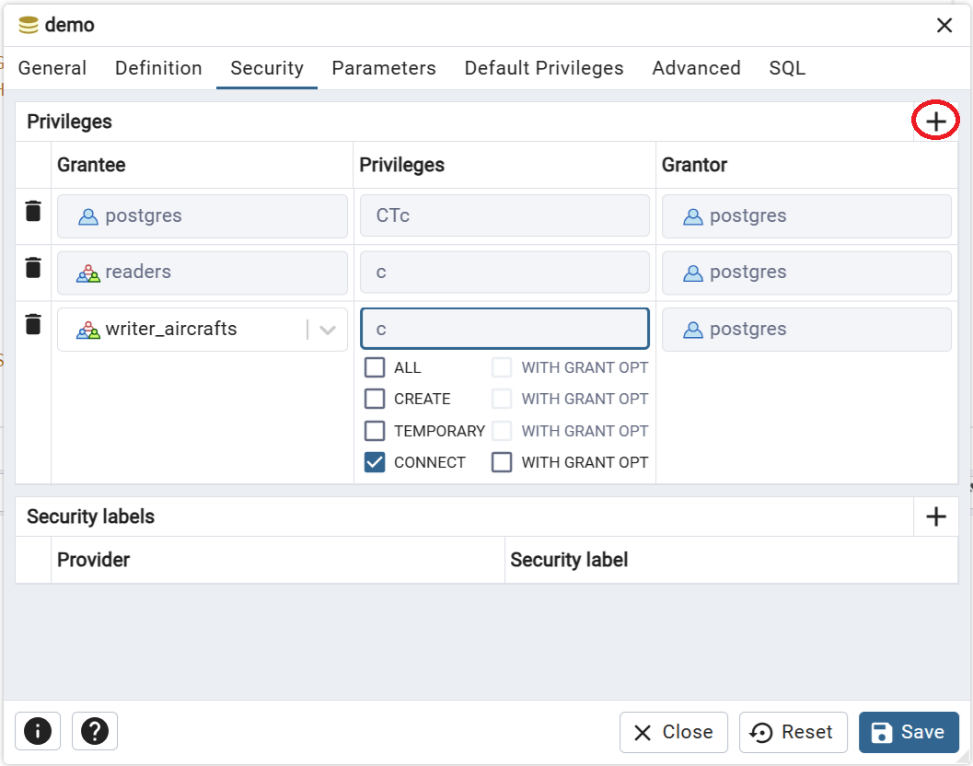


Рисунок 4.2.9

Убедимся, что мы сделали все правильно, открыв вкладку SQL  
 (рис. 4.2.10):

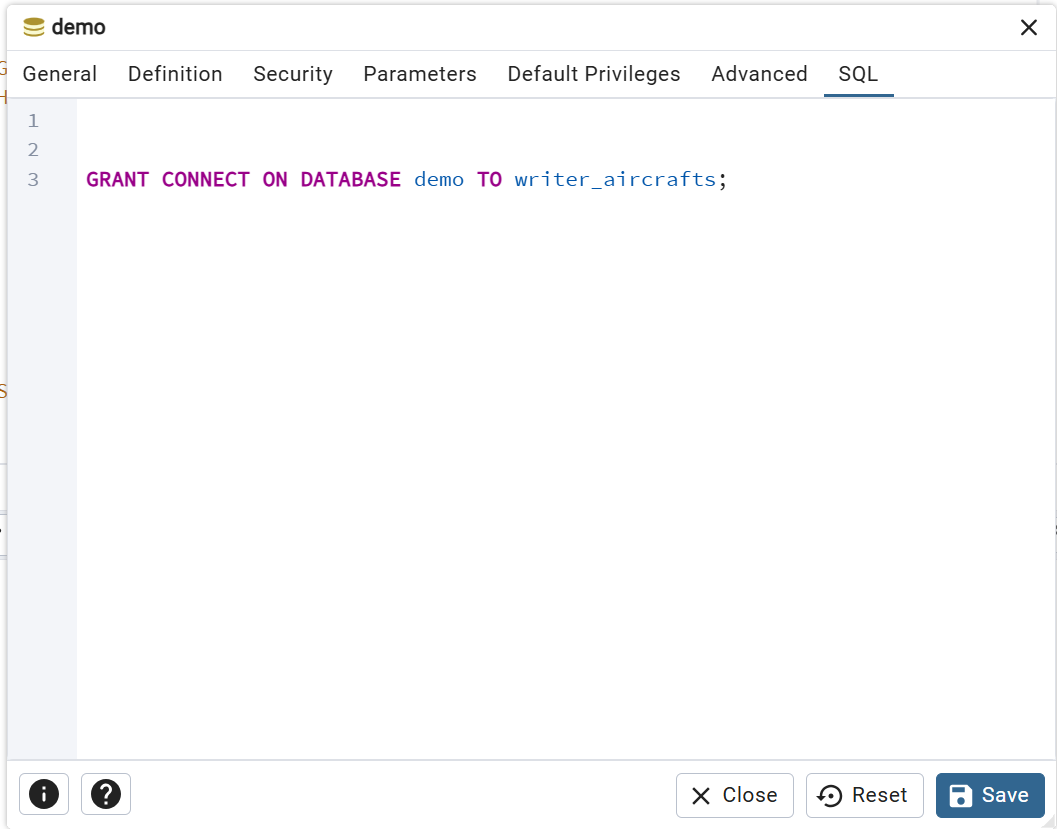


Рисунок 4.2.10

Сохраним. Проверим, что пользователь может подключиться к demo, проще всего это сделать, запустив утилиту psql (рис. 4.2.11):

Server [localhost]:

Database [postgres]: demo

Port [5432]:

Username [postgres]: fio2

Пароль пользователя fio2:

psql (16.8)

ПРЕДУПРЕЖДЕНИЕ: Кодовая страница консоли (866) отличается от основной

страницы Windows (1251).

8-битовые (русские) символы могут отображаться некорректно.

Подробнее об этом смотрите документацию psql, раздел

"Notes for Windows users".

Введите "help", чтобы получить справку.

demo=>

Рисунок 4.2.11

4.3. Задание прав на уровне схемы

Рассмотрим команду предоставление прав на схему:

GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }

ON SCHEMA *имя\_схемы* [, ...]

TO *указание\_роли* [, ...] [ WITH GRANT OPTION ]

**Параметры**

CREATE – означает, что пользователь может создавать в схеме объекты.

USAGE –означает, что пользователю разрешено войти в схему.

От имени суперпользователя дадим привилегию USAGE (рис. 4.3.1):

Пользователь postgres

demo=# GRANT USAGE ON SCHEMA bookings TO readers;

GRANT

Рисунок 4.3.1

Но это еще не значит, что пользователь сможет что-то из находящего в этой схеме использовать; соответствующие права еще не предоставлены. Пока что пользователю разрешено только просматривать эту схему в системном каталоге (рис. 4.3.2):

Пользователь fio1

demo=> SELECT \* FROM bookings.aircrafts\_data;

ОШИБКА: нет доступа к таблице aircrafts\_data

demo=> \dt

Список отношений

Схема | Имя | Тип | Владелец

----------+-----------------+---------+----------

bookings | aircrafts\_data | таблица | postgres

bookings | airports\_data | таблица | postgres

bookings | boarding\_passes | таблица | postgres

bookings | bookings | таблица | postgres

bookings | flights | таблица | postgres

bookings | seats | таблица | postgres

bookings | ticket\_flights | таблица | postgres

bookings | tickets | таблица | postgres

(8 строк)

Рисунок 4.3.2

Поскольку владелец таблиц postgres, нам необходимо выдать привилегии SELECT на таблицы для роли readers.

Проделаем тоже самое и для роли writer\_aircrafts, но используя pgAdmin. Последовательность действий аналогична, описанной выше для БД demo. Нажимаем ПКМ на узле bookings и выбираем из меню *Properties…*. Выбираем во вкладку Security и добавляем роль с привилегией USAGE   
(рис. 4.3.3):

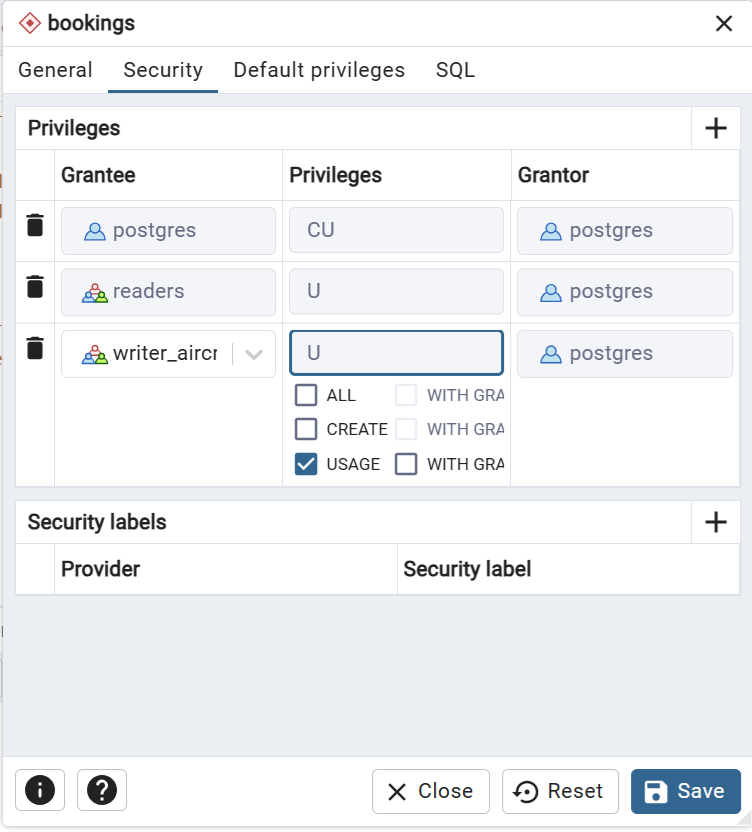


Рисунок 4.3.3

Сохраним. Теперь необходимо выдать привилегии для работы с таблицами.

4.4. Работа с таблицами

Для предоставления прав на таблицы надо использовать команду все той же командой GRANT:

GRANT {{SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER}

[, ...] | ALL [ PRIVILEGES ] }

ON { [ TABLE ] *имя\_таблицы* [, ...]

| ALL TABLES IN SCHEMA *имя\_схемы* [, ...] }

TO *указание\_роли* [, ...] [ WITH GRANT OPTION ]

Параметры

SELECT – позволяет выполнять SELECT для любого столбца или перечисленных столбцов в заданной таблице, представлении, или другом объекте табличного вида. Также позволяет выполнять COPY TO. Помимо этого, данное право требуется для обращения к существующим значениям столбцов в UPDATE, DELETE или MERGE.

INSERT – позволяет вставлять с помощью INSERT строки в заданную таблицу, представление и т. п. Также позволяет выполнять COPY FROM.

UPDATE – позволяет изменять с помощью UPDATE данные во всех, либо только перечисленных, столбцах в заданной таблице, представлении и т. п. (На практике для любой нетривиальной команды UPDATE потребуется и право SELECT, так как она должна обратиться к столбцам таблицы, чтобы определить, какие строки подлежат изменению, и/или вычислить новые значения столбцов.)

DELETE – позволяет удалять с помощью команды DELETE строки из таблицы, представления и т. п. (На практике для любой нетривиальной команды DELETE потребуется также право SELECT, так как она должна обратиться к колонкам таблицы, чтобы определить, какие строки подлежат удалению.)

TRUNCATE – позволяет опустошать таблицу с помощью TRUNCATE.

REFERENCES – позволяет создавать ограничение внешнего ключа, обращающееся к таблице или определённым столбцам таблицы.

TRIGGER – позволяет создавать триггер для таблицы, представления и т. п.

Для группы readers мы выдадим пава на просмотр всех таблиц в схеме bookings (рис. 4.4.1):

Пользователь postgres

demo=# GRANT SELECT ON ALL TABLES IN SCHEMA bookings TO readers;

GRANT

demo=#

Рисунок 4.4.1

Выполним SELECT с правами пользователя fio1 (рис. 4.4.2):

Пользователь fio1

demo=> SELECT \* FROM bookings.aircrafts\_data;

aircraft\_code | model | range

---------------+------------------------------------------------------------+-------

773 | {"en": "Boeing 777-300", "ru": "Боинг 777-300"} | 11100

763 | {"en": "Boeing 767-300", "ru": "Боинг 767-300"} | 7900

SU9 | {"en": "Sukhoi Superjet-100", "ru": "Сухой Суперджет-100"} | 3000

320 | {"en": "Airbus A320-200", "ru": "Аэробус A320-200"} | 5700

321 | {"en": "Airbus A321-200", "ru": "Аэробус A321-200"} | 5600

319 | {"en": "Airbus A319-100", "ru": "Аэробус A319-100"} | 6700

733 | {"en": "Boeing 737-300", "ru": "Боинг 737-300"} | 4200

CN1 | {"en": "Cessna 208 Caravan", "ru": "Сессна 208 Караван"} | 1200

CR2 | {"en": "Bombardier CRJ-200", "ru": "Бомбардье CRJ-200"} | 2700

(9 строк)

demo=>

Рисунок 4.4.2

Аналогичным образом, можно создать групповые роли с правами, позволяющие модифицировать данные в таблицах. Предоставим привилегии на изменения таблицы aircrafts\_data для роли writer\_aircrafts. Вызовем команду контекстного меню *Properties…* для таблицы aircrafts\_data и во вкладке *Security* добавим роль, предоставив привилегии только на добавление и изменение записей (рис. 4.4.3):

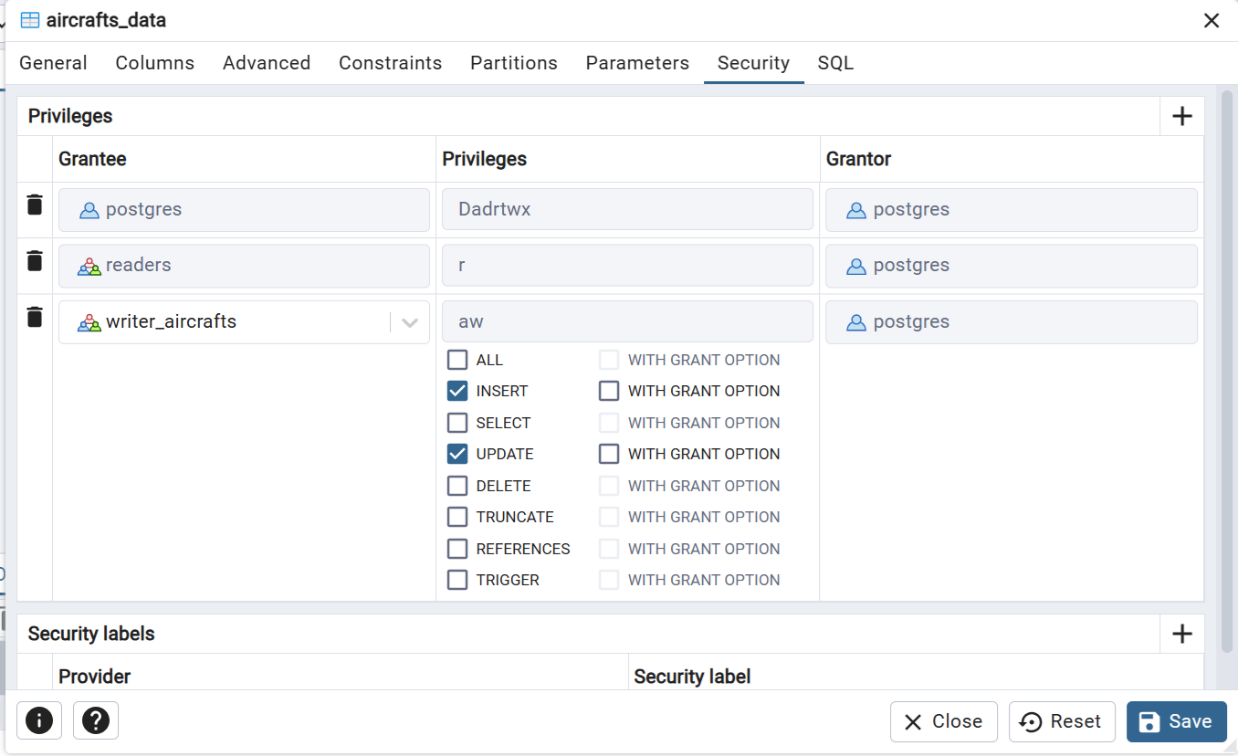


Рисунок 4.4.3

Войдем во вкладку и полюбопытствуем, какой скрипт нам приготовили (рис. 4.4.4):

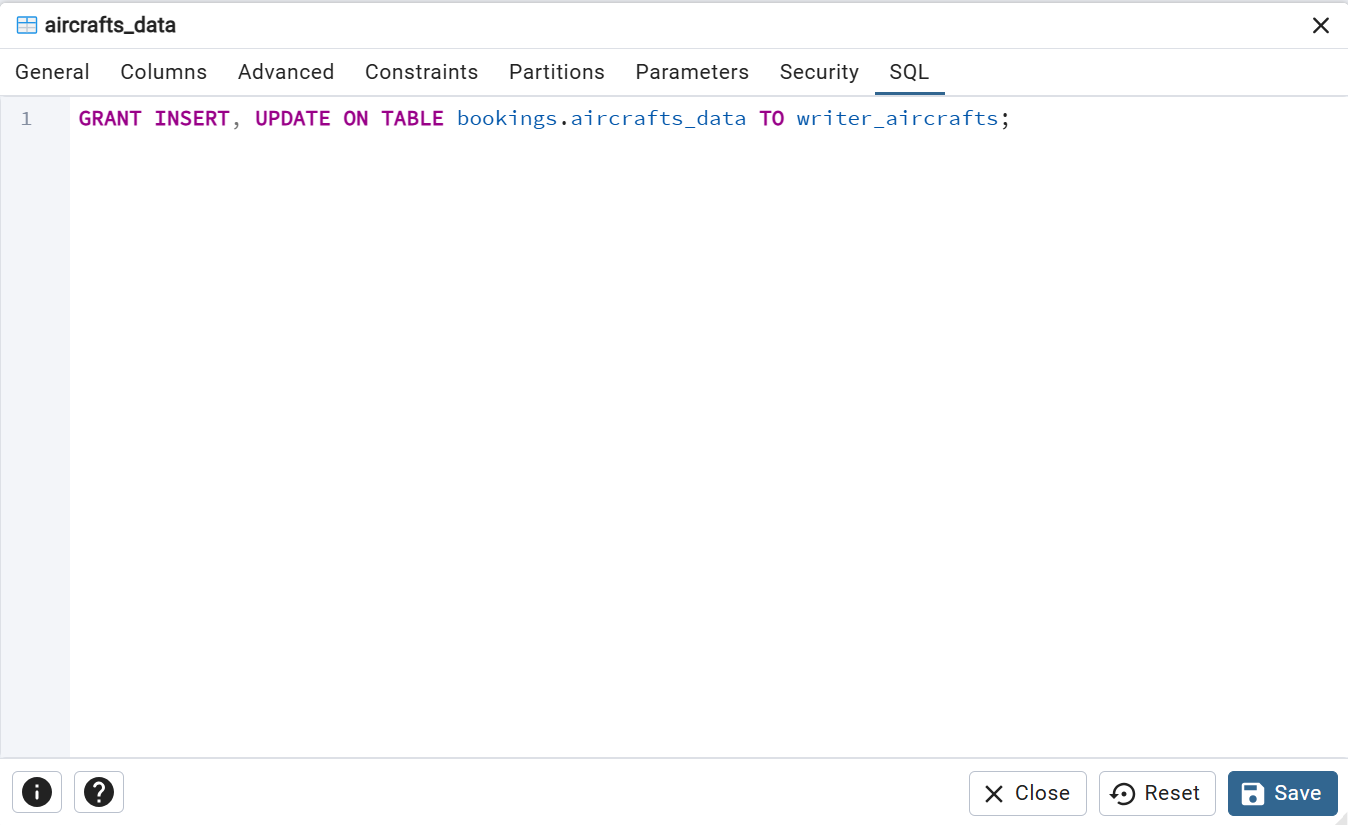


Рисунок 4.4.4

Теперь можем сохранить. Откроем psql или pgAdmin, как пользователь fio2 и добавим строку (рис. 14.5):

Server [localhost]:

Database [postgres]: demo

Port [5432]:

Username [postgres]: fio2

Пароль пользователя fio2:

psql (16.8)

ПРЕДУПРЕЖДЕНИЕ: Кодовая страница консоли (866) отличается от основной

страницы Windows (1251).

8-битовые (русские) символы могут отображаться некорректно.

Подробнее об этом смотрите документацию psql, раздел

"Notes for Windows users".

Введите "help", чтобы получить справку.

demo=> \! chcp 1251

Текущая кодовая страница: 1251

^

demo=> INSERT INTO bookings.aircrafts\_data ( aircraft\_code, model, range)   
VALUES('WWW', '{"en": "Sukhoi Superjet-200", "ru": "Сухой Суперджет-200" }', 7000);

INSERT 0 1

demo=>

Рисунок 4.4.5

Строка добавлена. При попытки удалить любую строку из таблицы aircrafts\_data мы получим сообщение об отсутствия доступа.

Теперь посмотрим на содержание таблицы и следующей строке наберем SELECT (рис. 4.4.6):

Пользователь fio2

demo=> SELECT \* FROM bookings.aircrafts\_data;

ОШИБКА: нет доступа к таблице aircrafts\_data

demo=>

Рисунок 4.4.6

Результат ожидаемый, нет прав. Вместо того чтобы предоставлять привилегии на SELECT для роли writer\_aircrafts, включим ее в роль readers, тем самым разрешим выполнять выборку из всех таблиц, т.к. привилегия SELECT наследуется (рис. 4.4.7):

Пользователь postgres

demo=# ALTER GROUP readers add USER writer\_aircrafts;

ALTER ROLE

demo=#

Рисунок 4.4.7

Повторим SELECT (рис. 4.4.8):

Пользователь fio2

demo=> SELECT \* FROM bookings.aircrafts\_data;

aircraft\_code | model | range

---------------+------------------------------------------------------------+------

773 | {"en": "Boeing 777-300", "ru": "Боинг 777-300"} | 11100

763 | {"en": "Boeing 767-300", "ru": "Боинг 767-300"} | 7900

SU9 | {"en": "Sukhoi Superjet-100", "ru": "Сухой Суперджет-100"} | 3000

320 | {"en": "Airbus A320-200", "ru": "Аэробус A320-200"} | 5700

321 | {"en": "Airbus A321-200", "ru": "Аэробус A321-200"} | 5600

319 | {"en": "Airbus A319-100", "ru": "Аэробус A319-100"} | 6700

733 | {"en": "Boeing 737-300", "ru": "Боинг 737-300"} | 4200

CN1 | {"en": "Cessna 208 Caravan", "ru": "Сессна 208 Караван"} | 1200

CR2 | {"en": "Bombardier CRJ-200", "ru": "Бомбардье CRJ-200"} | 2700

WWW | {"en": "Sukhoi Superjet-200", "ru": "Сухой Суперджет-200"} | 7000

(10 строк)

demo=>

Рисунок 4.4.8

Последний пример является демонстрацией удобства использования групповых ролей.

4.5. Примечание

Мы не рассмотрели возможность в pgAdmin выдавать привилегии ролям сразу для нескольких объектов БД. Это можно сделать, выбрав из контекстного меню для БД или схемы команду *Grant Wizard…*. Разобраться, как это работает, в силу изложенного выше материала, не составит труда и оставляется для самостоятельной проработки.

# 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 групповые роли: admin, reader и две роли writer. Каждая роль должна содержать по пользователю.
  + Роль члены роли admin должны обладать привилегиями суперпользователя. От имени суперпользователя из роли admin создать БД, схему и две таблицы.
  + Первая таблица (таблица 1) должна содержать id с типом serial. Вторая таблица (таблица 2) должна содержать id с типом identity.
  + Групповая роль reader должна обладать привилегиями чтения данных из таблиц БД.
  + Первая групповая роль writer1 должна иметь привилегии на чтение и изменения таблицы 1.
  + Вторая групповая роль writer2 должна иметь привилегии на чтение и изменения таблицы 2.
  + От имени пользователей из ролей writer добавьте, измените и удалите данные в соответствующих таблицах.
  + Привести скрипты для каждой операции по созданию ролей и данных. Объяснить принципиальную разницу в типах данных serial и identity.

# 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.
* Какие бывают типы триггеров?
* Когда срабатывает триггер?
* В каком порядке срабатывают триггеры?
* 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?
* Продемонстрировать откат транзакции при возникновении ошибок.
* Для чего нужны роли?
* 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?
* 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.

1. Версия 9.6 [↑](#footnote-ref-1)
2. ПКМ – сокращение «правая клавиша мыши». [↑](#footnote-ref-2)