Разграничение доступа

Обзор разграничения доступа

Авторы: Егор Рогов, Павел Лузанов

Обратная связь
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

Темы

Роли и атрибуты

Подключение к серверу

Привилегии

Политики защиты строк

Роли и атрибуты

Роль — пользователь СУБД

роль не связана с пользователем ОС

Свойства роли определяются атрибутами

`LOGIN`	возможность подключения
`SUPERUSER`	суперпользователь
`CREATEDB`	возможность создавать базы данных
`CREATEROLE`	возможность создавать роли
`REPLICATION`	использование протокола репликации

и другие

Роль — это пользователь СУБД. (Роль также может выступать в качестве *группы* пользователей, но об этом будет сказано позже.)

Формально роли никак не связаны с пользователями операционной системы, хотя многие программы это предполагают, выбирая значения по умолчанию. Например, psql, запущенный от имени пользователя ОС student, автоматически использует одноименную роль student (если в ключах утилиты не указана какая-либо другая роль).

При создании кластера определяется одна начальная роль, имеющая суперпользовательский доступ (обычно она называется postgres). В дальнейшем роли можно создавать, изменять и удалять.

https://postgrespro.ru/docs/postgresql/12/database-roles

Роль обладает некоторыми *атрибутами*, определяющими ее общие особенности и права (не связанные с правами доступа к объектам).

Обычно атрибуты имеют два варианта, например, CREATEDB (дает право на создание БД) и NOCREATEDB (не дает такого права). Как правило, по умолчанию выбирается ограничивающий вариант.

Если у роли нет атрибута LOGIN, она не сможет подключиться к серверу. (Такие роли тоже имеют смысл в качестве групповых.)

В таблице перечислены лишь некоторые из атрибутов. Атрибуты INHERIT и BYPASSRLS рассматривается чуть дальше в этой теме.

https://postgrespro.ru/docs/postgresql/12/role-attributes

https://postgrespro.ru/docs/postgresql/12/sql-createrole

Роли и атрибуты

Создадим роль для пользователя Алисы. В команде указаны два атрибута.

В этой теме нам важно, от имени какой роли выполняются команды, поэтому имя текущей роли вынесено в приглашение.

```
student=# CREATE ROLE alice LOGIN PASSWORD 'alicepass';
```

```
CREATE ROLE
```

Список ролей можно узнать командой:

```
student=# \du
```

```
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 alice     |                                                            | {}
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 student   | Superuser                                                  | {}
```

Обратите внимание, что роль student является суперпользователем. Поэтому до сих пор мы не задумывались о разграничении доступа.

Создадим и базу данных:

```
student=# CREATE DATABASE access_overview;
```

```
CREATE DATABASE
```

Подключение

1. Строки pg_hba.conf просматриваются сверху вниз
2. Выбирается первая запись, которой соответствуют параметры подключения (тип, база, пользователь и адрес)

```
# TYPE          DATABASE        USER            ADDRESS                 METHOD
local           all             postgres                                peer
local           all             all                                     peer
host            all             all             127.0.0.1/32            md5
host            all             all             ::1/128                 md5
```

local — сокет
host — TCP/IP

all — любая БД
имя БД

all — любая роль
имя роли

listen_addresses

all — любой IP
IP/маска
доменное имя

Для каждого нового клиента сервер определяет, надо ли разрешить подключение к базе данных. Настройки подключения определяются в конфигурационном файле pg_hba.conf («host-based authentication»). Как и с основным конфигурационным файлом postgresql.conf, изменения вступают в силу только после перечитывания файла сервером (SELECT pg_reload_conf() в SQL, либо pg_ctl reload из операционной системы).

При появлении нового клиента сервер просматривает конфигурационный файл сверху вниз в поисках строки, подходящей к запрашиваемому клиентом подключению. Соответствие определяется по четырем полям: типу подключения, имени БД, имени пользователя и IP-адресу.

Ниже перечислены только самые основные возможности.

Подключение — local (unix-сокеты, не доступно в Windows) или host (подключение по протоколу TCP/IP).

База данных – ключевое слово all (соответствует любой БД) или имя конкретной базы данных.

Пользователь — all или имя конкретной роли.

Адрес — all, конкретный IP-адрес с маской или доменное имя. Не указывается для типа local. По умолчанию PostgreSQL слушает входящие соединения только с localhost; обычно параметр listen_addresses ставят в значение «*» (слушать все интерфейсы) и дальше регулируют доступ средствами pg_hba.conf.

https://postgrespro.ru/docs/postgresql/12/client-authentication

Подключение

3. Выполняется аутентификация указанным методом
4. Удачно — доступ разрешается, иначе — запрещается
   (если не подошла ни одна запись — доступ запрещается)

```
# TYPE          DATABASE      USER              ADDRESS             METHOD
local           all           postgres                              peer
local           all           all                                   peer
host            all           all               127.0.0.1/32        md5
host            all           all               ::1/128             md5
```

trust — верить
reject — отказать
scram-sha-256 и md5 — запросить пароль
peer — спросить ОС

Когда в файле найдена подходящая строка, выполняется аутентификация указанным в этой строке методом, а также проверяется наличие атрибута LOGIN и привилегии CONNECT. Если результат проверки успешен, то подключение разрешается, иначе — запрещается (другие строки при этом уже не рассматриваются).

Если ни одна из строк не подошла, то доступ также запрещается.

Таким образом, записи в файле должны идти сверху вниз от частного к общему.

Существует множество методов аутентификации:

https://postgrespro.ru/docs/postgresql/12/auth-methods

Ниже перечислены только самые основные.

Метод trust безусловно разрешает подключение. Если вопросы безопасности не важны, можно указать «все all» и метод trust — тогда будут разрешены все подключения.

Метод reject наоборот, безусловно запрещает подключение.

Наиболее распространены методы md5 и более надежный scram-sha-256, которые запрашивают у пользователя пароль и проверяют его соответствие паролю, который хранится в системном каталоге кластера.

Метод peer запрашивает имя пользователя у операционной системы и разрешает подключение, если имя пользователя ОС и пользователя БД совпадают (можно установить и другие соответствия имен).

Парольная аутентификация

На сервере

- пароль устанавливается при создании роли или позже
- пользователю без пароля будет отказано в доступе
- пароль хранится в системном каталоге pg_authid

Ввод пароля на клиенте

- вручную
- из переменной окружения PGPASSWORD
- из файла ~/.pgpass (строки в формате *узел:порт:база:роль:пароль*)

Если используется аутентификация по паролю, для роли должен быть указан эталонный пароль, иначе в доступе будет отказано.

Пароль хранится в системном каталоге в таблице pg_authid.

Пользователь может вводить пароль вручную, а может автоматизировать ввод. Для этого есть две возможности.

Во-первых, можно задать пароль в переменной окружения PGPASSWORD на клиенте. Однако это неудобно, если приходится подключаться к нескольким базам, и не рекомендуется из соображений безопасности.

Во-вторых, можно задать пароли в файле ~/.pgpass на клиенте.
К файлу должен иметь доступ только владелец, иначе PostgreSQL проигнорирует его.

Подключение

Чтобы роль смогла подключиться к базе данных, она должна иметь не только атрибут LOGIN, но и разрешение в файле pg_hba.conf. Прочитать файл можно прямо из SQL:

```
student=# SELECT type, database, user_name, address, auth_method
FROM pg_hba_file_rules();
```

```
 type  |   database    | user_name  |  address  | auth_method
-------+---------------+------------+-----------+-------------
 local | {all}         | {postgres} |           | peer
 local | {all}         | {all}      |           | peer
 host  | {all}         | {all}      | 127.0.0.1 | md5
 host  | {all}         | {all}      | ::1       | md5
 local | {replication} | {all}      |           | peer
 host  | {replication} | {all}      | 127.0.0.1 | md5
 host  | {replication} | {all}      | ::1       | md5
(7 rows)
```

(В зависимости от сборки содержимое файла может отличаться.)

Мы будем использовать подключение по TCP/IP (host). Такому подключению соответствует третья строка выборки. Она предполагает аутентификацию по паролю.

Роль alice была создана с паролем, но вообще его можно изменить в любой момент:

```
student=# ALTER ROLE alice PASSWORD 'alicepass';
```

```
ALTER ROLE
```

Попробуем подключиться, указав в строке подключения всю необходимую информацию:

```
student$ psql "host=localhost user=alice dbname=access_overview password=alicepass"
```

```
alice=> \conninfo
```

```
You are connected to database "access_overview" as user "alice" on host "localhost" (address "127.0.0.1") at port "5432".
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
```

Получилось!

Привилегии

Привилегии определяют права доступа ролей к объектам

Таблицы

`SELECT`	чтение данных	можно на уровне столбцов
`INSERT`	вставка строк	
`UPDATE`	изменение строк	
`REFERENCES`	внешний ключ	
`DELETE`	удаление строк	
`TRUNCATE`	опустошение таблицы	
`TRIGGER`	создание триггеров	

Представления

`SELECT`	чтение данных
`TRIGGER`	создание триггеров

Привилегии определяются на пересечении объектов кластера и ролей. Они ограничивают действия, доступные для ролей в отношении этих объектов.

Список возможных привилегий отличается для объектов различных типов. Привилегии для основных объектов приведены на этом и следующем слайдах.

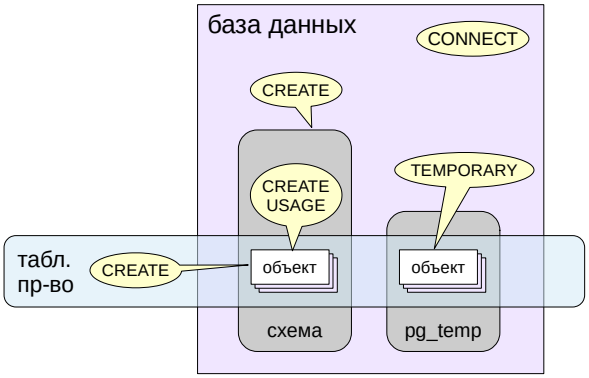
Больше всего привилегий определено для таблиц. Некоторые из них можно определить не только для всей таблицы, но и для отдельных столбцов.

https://postgrespro.ru/docs/postgresql/12/ddl-priv

https://postgrespro.ru/docs/postgresql/12/sql-grant

Привилегии

Табличные пространства,
базы данных, схемы



Последовательности

SELECT	currval		
UPDATE		nextval	setval
USAGE	currval	nextval	

Возможно, несколько неожиданный набор привилегий имеют последовательности. Выбирая нужные, можно разрешить или запретить доступ к трем управляющим функциям.

Для табличных пространств есть привилегия CREATE, разрешающая создание объектов в этом пространстве.

Для баз данных привилегия CREATE разрешает создавать схемы в этой БД, а для схемы привилегия CREATE разрешает создавать объекты в этой схеме.

Поскольку точное имя схемы для временных объектов заранее неизвестно, привилегия на создание временных таблиц вынесено на уровень БД (TEMPORARY).

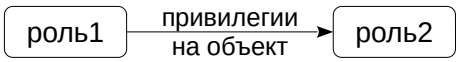
Привилегия USAGE схемы разрешает обращаться к объектам в этой схеме.

Привилегия CONNECT базы данных разрешает подключение к этой БД.

Управление привилегиями

Выдача привилегий

```
роль1: GRANT привилегии ON объект TO роль2;
```



Отзыв привилегий

```
роль1: REVOKE привилегии ON объект FROM роль2;
```


Право выдачи и отзыва привилегий на объект имеет владелец этого объекта (и суперпользователь).

Синтаксис команд GRANT и REVOKE достаточно сложен и позволяет указывать как отдельные, так и все возможные привилегии; как отдельные объекты, так и группы объектов, входящие в определенные схемы и т. п.

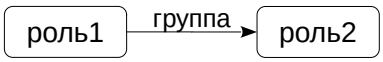
https://postgrespro.ru/docs/postgresql/12/sql-grant

https://postgrespro.ru/docs/postgresql/12/sql-revoke

Групповые роли

Включение роли в группу

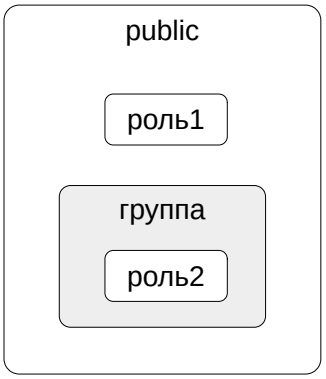
```
роль1: GRANT группа TO роль2;
```



псевдороль public неявно включает в себя все остальные роли

Исключение роли из группы

```
роль1: REVOKE группа FROM роль2;
```



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 упрощает администрирование и управление доступом.

Существует псевдороль public, которая неявно включает в себя все остальные роли. Если выдать какие-либо привилегии роли public, эти привилегии получат вообще все роли.

Правом на включение и исключение других ролей в данную роль обладают:

- сама эта роль,

- роль с атрибутом SUPERUSER (суперпользователь)

- роль с атрибутом CREATEROLE.

https://postgrespro.ru/docs/postgresql/12/role-membership

Суперпользователи

Кто входит в категорию

роли с атрибутом SUPERUSER

Права

полный доступ ко всем объектам — проверки не выполняются

В целом можно сказать, что доступ роли к объекту определяется привилегиями. Но имеет смысл выделить три категории ролей и рассмотреть их по отдельности.

Проще всего с ролями с атрибутом суперпользователя. Такие роли могут делать все, что угодно — для них проверки разграничения доступа не выполняются.

Владелец объекта

Кто входит в категорию

- изначально — роль, создавшая объект (можно сменить)
- а также роли, включенные в роль владельца

Права

- изначально все привилегии для объекта (можно отозвать)
- действия со своими объектами, не регламентируемые привилегиями, например: удаление, выдача и отзыв привилегий и т. п.

У каждого объекта есть роль, владеющая этим объектом («владелец»). Изначально это роль, создавшая объект, хотя потом владельца можно сменить. Неочевидный момент: владельцем считается не только сама роль-владелец, но и любая другая роль, включенная в нее.

Владелец объекта сразу получает полный набор привилегий для этого объекта.

В принципе, эти привилегии можно отозвать, но владелец объекта обладает также неотъемлемым правом совершать действия, не регламентируемые привилегиями. В частности, он может выдавать и отзывать привилегии (в том числе и себе самому), удалять объект и т. п.

Остальные роли

Кто входит в категорию

все остальные (не суперпользователи и не владельцы)

Права

доступ в рамках выданных привилегий

обычно наследуют привилегии групповых ролей
(но атрибут NOINHERIT требует явного переключения роли)

Наконец, все остальные роли имеют доступ к объекту только в рамках выданных им привилегий. В том числе учитываются и привилегии групповых ролей, в которые эти роли входят (включая псевдороль public, в которую неявно включены все остальные роли).

Обычно роль сразу обладает всеми «групповыми» полномочиями. Это поведение можно изменить, указав роли атрибут NOINHERIT — тогда, чтобы воспользоваться привилегиями групповых ролей, надо будет явно переключиться с помощью SET ROLE.

Чтобы проверить, есть ли у роли необходимая привилегия в отношении некоторого объекта, можно воспользоваться функциями has_*_privilege:

https://postgrespro.ru/docs/postgresql/12/functions-info

Выданные привилегии удобно показывают команды psql, описывающие объект (см. раздаточный материал по системному каталогу).

Привилегии

Алиса смогла подключиться к базе данных. Теперь она хочет создать для себя схему и несколько объектов в ней.

```
alice=> CREATE SCHEMA alice;

ERROR:  permission denied for database access_overview
```

В чем проблема?

У Алисы нет привилегии для создания схем в БД. Выдадим ее:

```
student=# GRANT CREATE ON DATABASE access_overview TO alice;

GRANT
```

Пробуем еще раз:

```
alice=> CREATE SCHEMA alice;

CREATE SCHEMA
```

Теперь, поскольку Алиса является владельцем своей схемы, она имеет все привилегии на нее и может создавать в ней любые объекты. По умолчанию будет использоваться именно эта схема:

```
alice=> SELECT current_schemas(true);

      current_schemas
---------------------------
 {pg_catalog,alice,public}
(1 row)
```

Создадим две таблицы.

```
alice=> CREATE TABLE t1(n numeric);

CREATE TABLE

alice=> INSERT INTO t1 VALUES (1);

INSERT 0 1

alice=> CREATE TABLE t2(n numeric, who text DEFAULT current_user);

CREATE TABLE

alice=> INSERT INTO t2(n) VALUES (1);

INSERT 0 1
```

Теперь создадим другую роль для пользователя Боба, который будет обращаться к объектам, принадлежащим Алисе.

```
student=# CREATE ROLE bob LOGIN PASSWORD 'bobpass';

CREATE ROLE

student$ psql "host=localhost user=bob dbname=access_overview password=bobpass"
```

Боб попробует обратиться к таблице t1.

```
bob=> SELECT * FROM alice.t1;

ERROR:  permission denied for schema alice
LINE 1: SELECT * FROM alice.t1;
                       ^
```

В чем причина ошибки?

Нет доступа к схеме, так как Боб не суперпользователь и не владелец.

Проверить права на доступ к схеме можно так (столбец Access privileges):

```
alice=> \dn+
```

```
                          List of schemas
  Name  |  Owner   |  Access privileges   |      Description
--------+----------+----------------------+------------------------
 alice  | alice    |                      |
 public | postgres | postgres=UC/postgres+| standard public schema
        |          | =UC/postgres         |
(2 rows)
```

Привилегии отображаются в формате: роль=привилегии/кем_предоставлены.

Каждая привилегия кодируется одним символом, в частности для схем:

- U = usage;
- C = create.

Если имя роли опущено (как в последней строке), то имеется в виду псевдороль public.

Если же опущено все поле (как в первой строке), то имеются в виду привилегии владельца по умолчанию. Здесь alice имеет обе доступные привилегии на собственную схему.

Предоставим доступ к схеме для Боба. Это может сделать Алиса, как владелец.

```
alice=> GRANT CREATE, USAGE ON SCHEMA alice TO bob;
```

```
GRANT
```

Боб снова пробует обратиться к таблице:

```
bob=> SELECT * FROM alice.t1;
```

```
ERROR:  permission denied for table t1
```

В чем причина ошибки?

На этот раз у Боба есть доступ к схеме, но нет доступа к самой таблице. Вот как проверить доступ:

```
alice=> \dp alice.t1
```

```
                            Access privileges
 Schema | Name | Type  | Access privileges | Column privileges | Policies
--------+------+-------+-------------------+-------------------+----------
 alice  | t1   | table |                   |                   |
(1 row)
```

И снова видим пустое поле: доступ есть только у владельца, то есть Алисы.

Алиса предоставляет Бобу доступ на чтение и изменение:

```
alice=> GRANT SELECT,UPDATE ON alice.t1 TO bob;
```

```
GRANT
```

А для второй таблицы — доступ на вставку и чтение одного столбца:

```
alice=> GRANT SELECT(n),INSERT ON alice.t2 TO bob;
```

```
GRANT
```

Посмотрим, как изменились привилегии:

```
alice=> \dp alice.*
```

```
                             Access privileges
 Schema | Name | Type  |  Access privileges  | Column privileges | Policies
--------+------+-------+---------------------+-------------------+----------
 alice  | t1   | table | alice=arwdDxt/alice+|                   |
        |      |       | bob=rw/alice        |                   |
 alice  | t2   | table | alice=arwdDxt/alice+| n:               +|
        |      |       | bob=a/alice         |   bob=r/alice     |
(2 rows)
```

Теперь пустое поле «проявилась», и мы видим, что в нем находится полный перечень привилегий. Вот обозначения, не все вполне очевидные:

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

Привилегии для столбцов отображаются отдельно (столбец Column privileges).

На этот раз попытки Боба увенчаются успехом. Чтобы не указывать каждый раз имя схемы, Боб добавляет его к своему пути поиска.

```
bob=> SET search_path = public, alice;
SET
bob=> UPDATE t1 SET n = n + 1;
UPDATE 1
bob=> SELECT * FROM t1;
 n
---
 2
(1 row)
```

Но другие операции по-прежнему запрещены:

```
bob=> DELETE FROM t1;
ERROR:  permission denied for table t1
```

И вторая таблица:

```
bob=> INSERT INTO t2(n) VALUES (100);
INSERT 0 1
bob=> SELECT n FROM t2;
  n
-----
   1
 100
(2 rows)
```

А чтение другого столбца запрещено:

```
bob=> SELECT * FROM t2;
ERROR:  permission denied for table t2
```

Подпрограммы

Единственная привилегия для функций и процедур

EXECUTE	выполнение

Характеристики безопасности

SECURITY INVOKER	выполняется с правами вызывающего (по умолчанию)
SECURITY DEFINER	выполняется с правами владельца

Для функций и процедур есть единственная привилегия EXECUTE, разрешающая выполнение этой подпрограммы.

Тонкий момент связан с тем, от имени какого пользователя будет выполняться подпрограмма. Если подпрограмма объявлена как SECURITY INVOKER (по умолчанию), она выполняется с правами вызывающего пользователя. В этом случае операторы внутри подпрограммы смогут обращаться только к тем объектам, к которым у вызывающего пользователя есть доступ.

Если же указать фразу SECURITY DEFINER, подпрограмма работает с правами создавшего ее пользователя. Это способ позволить другим пользователям выполнять определенные действия над объектами, к которым у них нет непосредственного доступа.

https://postgrespro.ru/docs/postgresql/12/sql-createfunction

https://postgrespro.ru/docs/postgresql/12/sql-createprocedure

Привилегии по умолчанию

Привилегии псевдороли public

- подключение к любой базе данных
- доступ к схеме public и создание в ней объектов
- доступ к системному каталогу
- выполнение любых подпрограмм
- привилегии выдаются автоматически для каждого нового объекта

Настраиваемые привилегии по умолчанию

- возможность дополнительно выдать или отозвать привилегии для только что созданного объекта

Как уже говорилось, псевдороль public включает в себя все остальные роли, которые, таким образом, пользуются всеми привилегиями, выданными для public.

При этом public имеет довольно широкий спектр привилегий по умолчанию. В частности:

- право подключения к любой базе данных (именно поэтому роль alice смогла подключиться к базе данных несмотря на то, что привилегия CONNECT не выдавалась ей явно);

- доступ к системному каталогу и схеме public;

- и к тому же — право выполнения любых подпрограмм.

Это, с одной стороны, позволяет комфортно работать, не задумываясь о привилегиях, но с другой, создает определенные сложности, если разграничение доступа действительно необходимо.

Описанные выше привилегии появляются у public автоматически при создании новых объектов. То есть недостаточно просто отозвать у public привилегию EXECUTE: как только появится новая подпрограмма, public немедленно получает привилегию на ее выполнение.

Существует специальный механизм «привилегий по умолчанию», который позволяет автоматически выдавать необходимые привилегии при создании нового объекта. Этот механизм можно использовать и для отзыва права выполнения функций у псевдороли public.

https://postgrespro.ru/docs/postgresql/12/sql-alterdefaultprivileges

Подпрограммы и привилегии по умолчанию

Алиса создает функцию:

```
alice=> CREATE FUNCTION foo() RETURNS SETOF t2
AS $$
SELECT * FROM t2;
$$ LANGUAGE sql STABLE;
```

```
CREATE FUNCTION
```

Сможет ли Боб вызвать ее, если Алиса не выдала ему привилегию EXECUTE?

```
bob=> SELECT foo();
```

```
ERROR:  permission denied for table t2
CONTEXT:  SQL function "foo" statement 1
```

Вызвать — да, но Боб не сможет получить доступ к объектам, на которые ему не выданы привилегии.

Если же Боб создаст свою таблицу t2 в схеме public, функция будет работать для обоих пользователей — с разными таблицами, поскольку у Алисы и Баба разные пути поиска:

```
bob=> CREATE TABLE t2(n numeric, who text DEFAULT current_user);
```

```
CREATE TABLE
```

```
bob=> INSERT INTO t2(n) VALUES (42);
```

```
INSERT 0 1
```

```
bob=> SELECT foo();
```

```
   foo
----------
 (42,bob)
(1 row)
```

```
alice=> SELECT foo();
```

```
    foo
-----------
 (1,alice)
 (100,bob)
(2 rows)
```

Другой доступный вариант — объявить функцию, как работающую с правами создавшего (SECURITY DEFINER):

```
alice=> ALTER FUNCTION foo() SECURITY DEFINER;
```

```
ALTER FUNCTION
```

В этом случае функция работает с правами создавшей роли, независимо от того, кто ее вызывает.

Боб удаляет свою таблицу...

```
bob=> DROP TABLE t2;
```

```
DROP TABLE
```

...и получает доступ к содержимому таблицы Алисы:

```
bob=> SELECT foo();
```

```
    foo
-----------
 (1,alice)
 (100,bob)
(2 rows)
```

В таком случае Алисе надо внимательно следить за выданными привилегиями. Скорее всего, потребуется отозвать EXECUTE у роли public и выдавать ее явно только нужным ролям.

```
alice=> REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA alice FROM public;
```

```
REVOKE
```

```
bob=> SELECT foo();
```

```
ERROR:  permission denied for function foo
```

Дело осложняется тем, что привилегия на выполнение автоматически выдается роли public на каждую вновь создаваемую функцию.

```
alice=> CREATE FUNCTION bar() RETURNS integer AS $$
SELECT 1;
$$ LANGUAGE sql IMMUTABLE SECURITY DEFINER;
```

```
CREATE FUNCTION
```

```
bob=> SELECT bar();
```

```
 bar
-----
   1
(1 row)
```

Однако этого можно избежать, изменив привилегии по умолчанию:

```
alice=> ALTER DEFAULT PRIVILEGES
FOR ROLE alice
REVOKE EXECUTE ON FUNCTIONS FROM public;
```

```
ALTER DEFAULT PRIVILEGES
```

```
alice=> \ddp
```

```
           Default access privileges
 Owner | Schema |   Type   | Access privileges
-------+--------+----------+-------------------
 alice |        | function | alice=X/alice
(1 row)
```

Теперь только Алиса получает привилегию на выполнение своих функций, а public — нет.

```
alice=> CREATE FUNCTION baz() RETURNS integer AS $$
SELECT 1;
$$ LANGUAGE sql IMMUTABLE SECURITY DEFINER;
```

```
CREATE FUNCTION
```

```
bob=> SELECT baz();
```

```
ERROR:  permission denied for function baz
```

Политики защиты строк

Дополнение к системе привилегий
для разграничения доступа к таблицам на уровне строк

Политика применяется

- к определенным ролям
- к определенным командам (SELECT, INSERT, UPDATE, DELETE)

Политика определяет условие доступности строки

- разрешительная: позволяет видеть строку, если условие выполнено
- ограничительная: запрещает видеть строку, если условие не выполнено
- отдельные условия (предикаты) для существующих и для новых строк

Привилегии позволяют разграничить доступ на уровне таблиц и столбцов, а политики защиты строк (row-level security) — на уровне строк.

Защита строк по умолчанию выключена. При необходимости ее надо включать явно для каждой таблицы.

Политики определяется для таблицы для набора команд (SELECT, INSERT, UPDATE, DELETE) и для определенных ролей. По сути, каждая из политик — это логическое условие (предикат), вычисляемое для каждой строки выборки. Если условие истинно — доступ к строке разрешается (достаточно, чтобы доступ позволила хотя бы одна *разрешительная* политика и не запретила ни одна *ограничительная* политика). В противном случае случае строка не будет видна.

Можно указывать разные предикаты для доступа к существующим строкам и для добавления новых строк (тогда, например, операция UPDATE сработает, только если оба предикаты будут истинны).

Предикаты вычисляются с правами вызывающего.

Политики защиты строк не действуют на владельца таблицы (как правило), на суперпользователей, на роли со специальным атрибутом BYPASSRLS и на ограничения целостности (уникальность, внешние ключи).

https://postgrespro.ru/docs/postgresql/12/ddl-rowsecurity

Политики защиты строк

Политики защиты позволяют разграничить доступ к таблице по строкам в зависимости от текущей роли.

```
alice=> SELECT * FROM t2;
```

```
  n  |  who
-----+-------
   1 | alice
 100 | bob
(2 rows)
```

Для примера сделаем так, чтобы роль, читающая таблицу, могла видеть только «свои» строки, в которых поле who содержит ее имя.

```
alice=> CREATE POLICY who_policy ON t2
USING (who = current_user);
```

```
CREATE POLICY
```

Чтобы защита начала работать, ее необходимо включить на уровне таблицы:

```
alice=> ALTER TABLE t2 ENABLE ROW LEVEL SECURITY;
```

```
ALTER TABLE
```

Теперь Боб видит только «свои» строки. Фактически, при выполнении запроса каждая строка проверяется на выполнение указанного в политике предиката.

```
bob=> SELECT n FROM t2;
```

```
  n
-----
 100
(1 row)
```

```
bob=> INSERT INTO t2(n) VALUES (101);
```

```
INSERT 0 1
```

```
bob=> SELECT n FROM t2;
```

```
  n
-----
 100
 101
(2 rows)
```

Политики защиты строк не действуют на суперпользователей и на роли с атрибутом BYPASSRLS. На владельца таблицы политики тоже обычно не действуют:

```
alice=> SELECT * FROM t2;
```

```
  n  |  who
-----+-------
   1 | alice
 100 | bob
 101 | bob
(3 rows)
```

Но Алиса может ограничить и сама себя:

```
alice=> ALTER TABLE t2 FORCE ROW LEVEL SECURITY;
```

```
ALTER TABLE
```

```
alice=> SELECT * FROM t2;
```

```
 n |  who
---+-------
 1 | alice
(1 row)
```

Итоги

Роли, привилегии и политики — гибкий механизм, позволяющий по-разному организовать работу

- можно легко разрешить все всем
- можно строго разграничить доступ, если это необходимо

При создании новых ролей надо позаботиться о возможности их подключения к серверу

Практика

1. Создайте две роли (пароль должен совпадать с именем):
   – employee — сотрудник магазина,
   – buyer — покупатель.

   Убедитесь, что созданные роли могут подключиться к БД.
2. Отзовите у роли public права выполнения всех функций и подключения к БД.
3. Разграничьте доступ таким образом, чтобы:
   – сотрудник мог только заказывать книги, а также добавлять авторов и книги,
   – покупатель мог только приобретать книги.

   Проверьте выполненные настройки в приложении.

1. Сотрудник — внутренний пользователь приложения, аутентификация выполняется на уровне СУБД.

Покупатель — внешний пользователь. В реальном интернет-магазине управление такими пользователями ложится на приложение, а все запросы поступают в СУБД от одной «обобщенной» роли (buyer). Идентификатор конкретного покупателя может передаваться как параметр (но в нашем приложении мы этого не делаем).

3. Вообще говоря, разграничение доступа должно быть заложено, в том числе, в приложение. В нашем учебном приложении разграничение не сделано специально: вместо этого на веб-странице можно явно выбрать роль, от имени которой пойдет запрос в СУБД. Это позволяет посмотреть, как поведет себя серверная часть при некорректной работе приложения.

Итак, пользователям нужно выдать:

- Право подключения к БД bookstore и доступ к схеме bookstore.

- Доступ к представлениям, к которым происходит непосредственное обращение.

- Доступ к функциям, которые вызываются как часть API. Если оставить функции SECURITY INVOKER, придется выдавать доступ и ко всем «нижележащим» объектам (таблицам, другим функциям). Однако удобнее просто объявить API-функции как SECURITY INVOKER.

Разумеется, ролям нужно выдать привилегии только на те объекты, доступ к которым у них должен быть.

1. Создание ролей

```
=> CREATE ROLE employee LOGIN PASSWORD 'employee';
```

```
CREATE ROLE
```

```
=> CREATE ROLE buyer LOGIN PASSWORD 'buyer';
```

```
CREATE ROLE
```

Настройки по умолчанию разрешают подключение с локального адреса по паролю. Нас это устраивает.

2. Привилегии public

У роли public надо отозвать лишние привилегии.

```
=> REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA bookstore FROM public;
```

```
REVOKE
```

```
=> REVOKE CONNECT ON DATABASE bookstore FROM public;
```

```
REVOKE
```

3. Разграничение доступа

Функции с правами создавшего.

```
=> ALTER FUNCTION get_catalog(text,text,boolean) SECURITY DEFINER;
```

```
ALTER FUNCTION
```

```
=> ALTER FUNCTION update_catalog() SECURITY DEFINER;
```

```
ALTER FUNCTION
```

```
=> ALTER FUNCTION add_author(text,text,text) SECURITY DEFINER;
```

```
ALTER FUNCTION
```

```
=> ALTER FUNCTION add_book(text,integer[]) SECURITY DEFINER;
```

```
ALTER FUNCTION
```

```
=> ALTER FUNCTION buy_book(integer) SECURITY DEFINER;
```

```
ALTER FUNCTION
```

```
=> ALTER FUNCTION book_name(integer,text,integer) SECURITY DEFINER;
```

```
ALTER FUNCTION
```

```
=> ALTER FUNCTION authors(books) SECURITY DEFINER;
```

```
ALTER FUNCTION
```

Привилегии покупателя: покупатель должен иметь доступ к поиску книг и их покупке.

```
=> GRANT CONNECT ON DATABASE bookstore TO buyer;
```

```
GRANT
```

```
=> GRANT USAGE ON SCHEMA bookstore TO buyer;
```

```
GRANT
```

```
=> GRANT EXECUTE ON FUNCTION get_catalog(text,text,boolean) TO buyer;
```

```
GRANT
```

```
=> GRANT EXECUTE ON FUNCTION buy_book(integer) TO buyer;
```

```
GRANT
```

Привилегии сотрудника: сотрудник должен иметь доступ к просмотру и добавлению книг и авторов, а также к каталогу для заказа книг.

```
=> GRANT CONNECT ON DATABASE bookstore TO employee;
```

```
GRANT
```

```
=> GRANT USAGE ON SCHEMA bookstore TO employee;
```

```
GRANT
=> GRANT SELECT,UPDATE(onhand_qty) ON catalog_v TO employee;
GRANT
=> GRANT SELECT ON authors_v TO employee;
GRANT
=> GRANT EXECUTE ON FUNCTION book_name(integer,text,integer) TO employee;
GRANT
=> GRANT EXECUTE ON FUNCTION authors(books) TO employee;
GRANT
=> GRANT EXECUTE ON FUNCTION author_name(text,text,text) TO employee;
GRANT
=> GRANT EXECUTE ON FUNCTION add_book(text,integer[]) TO employee;
GRANT
=> GRANT EXECUTE ON FUNCTION add_author(text,text,text) TO employee;
GRANT
```

Практика

Подпрограммы, объявленные как выполняющиеся с правами владельца (SECURITY DEFINER), могут использоваться, чтобы предоставить обычным пользователям доступ к возможностям, доступным только суперпользователю.

1. Создайте обычного непривилегированного пользователя и проверьте, что он не может изменять значение параметра *log_statement*.
2. Напишите подпрограмму для включения и выключения трассировки SQL-запросов так, чтобы созданная роль могла ей воспользоваться.

1. Вспомните демонстрацию к теме «PL/pgSQL. Отладка». В ней не возникало сложностей с установкой параметра, поскольку демонстрация выполнялась от имени роли student, которая является суперпользователем.

1. Создание роли и проверка

```
student=# CREATE DATABASE access_overview;
```

```
CREATE DATABASE
```

```
student=# \c access_overview
```

```
You are now connected to database "access_overview" as user "student".
```

```
student=# CREATE ROLE alice LOGIN PASSWORD 'alicepass';
```

```
CREATE ROLE
```

```
student$ psql "host=localhost user=alice dbname=access_overview password=alicepass"
```

Алиса не может изменить значение параметра:

```
alice=> SET log_statement = 'all';
```

```
ERROR:  permission denied to set parameter "log_statement"
```

2. Процедура для трассировки

От имени суперпользователя создаем процедуру для изменения параметра и объявляем ее SECURITY DEFINER:

```
student=# CREATE PROCEDURE trace(val boolean)
AS $$
SELECT set_config(
    'log_statement',
    CASE WHEN val THEN 'all' ELSE 'none' END,
    false /* is_local */
);
$$ LANGUAGE sql SECURITY DEFINER;
```

```
CREATE PROCEDURE
```

Отбираем права на выполнение у роли public...

```
student=# REVOKE EXECUTE ON PROCEDURE trace FROM public;
```

```
REVOKE
```

...и выдаем Алисе. Вместо того, чтобы выбирать между FUNCTION и PROCEDURE, почти все команды (кроме CREATE) позволяют использовать общее слово ROUTINE:

```
student=# GRANT EXECUTE ON ROUTINE trace TO alice;
```

```
GRANT
```

Теперь Алиса может включать и выключать трассировку, хотя и не имеет непосредственного доступа к параметру:

```
alice=> CALL trace(true);
```

```
CALL
```

```
alice=> SELECT 2*2;
```

```
 ?column?
----------
        4
(1 row)
```

```
alice=> CALL trace(false);
```

```
CALL
```

```
student$ tail -n 2 /var/log/postgresql/postgresql-12-main.log
```

```
2021-01-12 20:37:00.876 MSK [207979] alice@access_overview LOG:  statement: SELECT 2*2;
2021-01-12 20:37:00.978 MSK [207979] alice@access_overview LOG:  statement: CALL trace(false);
```