Data Base Administrator PostgreSQL

Ernesto Quiñones A.
ernesto@eqsoft.net

Sesión 4

Vistas y Vistas Materializadas
Transacciones
Usuarios y Roles
Administración Básica

Vistas

Una Vista (View) es una representación lógica (no física) de los datos de nuestra db, se construye aplicando una consulta a nuestra base de datos (query) y permanecerá como un objeto de la base de datos hasta que la eliminemos.

Las vista consumen recursos en nuestra base de datos, y cada vez que se invoca ejecutará la consulta con la cual se ha construido.

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name
[ ( column_name [, ...] ) ]
    [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Vistas

```
prueba4=# create table tablita (id serial, valores int, valores2
int);
CREATE TABLE

prueba4=# insert into tablita (valores, valores2) values
( generate_series(1,10000) * random(), 123 * random());
INSERT 0 10000

prueba4=# select count(distinct valores), count (distinct valores2)
from tablita;
 count | count
-------+-------
  5027 |   124
```

Vistas

```
prueba4=# create view vista1 as select * from tablita;
CREATE VIEW

prueba4=# \d vista1
      View "public.vista1"
  Column  |  Type   | Modifiers
----------+---------+-----------
 id       | integer |
 valores  | integer |
 valores2 | integer |

prueba4=# select count(distinct valores), count (distinct valores2)
from vista1;
 count | count
-------+-------
  5027 |   124
(1 row)
```

Vistas

Podemos crear una VISTA de registros filtrados:

```
prueba4=# create view vista2 as select * from tablita where valores2 = 123;
CREATE VIEW
prueba4=# select count(*) from vista2;
count
-------
   31
```

Sin embargo el tiempo de resolver nuestra consulta es exactamente igual.

```
prueba4=# explain analyze select * from tablita where valores2 = 123;
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------
Seq Scan on tablita  (cost=0.00..180.00 rows=81 width=12) (actual time=0.010..0.952 rows=31
loops=1)
  Filter: (valores2 = 123)
  Rows Removed by Filter: 9969

prueba4=# explain analyze select * from vista2;
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------
Seq Scan on tablita  (cost=0.00..180.00 rows=81 width=12) (actual time=0.014..0.999 rows=31
loops=1)
  Filter: (valores2 = 123)
  Rows Removed by Filter: 9969
```

Vistas

Si se crea un índice en la tabla este puede ser utilizado en las vistas que dependen de esa tabla, no se pueden crear índices sobre las vistas.

```
prueba4=# create index idx1 on tablita(valores2);
CREATE INDEX

prueba4=# explain analyze select * from tablita where valores2 = 123;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on tablita  (cost=4.91..63.58 rows=81 width=12) (actual time=0.030..0.054 rows=31
loops=1)
  Recheck Cond: (valores2 = 123)
  Heap Blocks: exact=26
  ->  Bitmap Index Scan on idx1  (cost=0.00..4.89 rows=81 width=0) (actual time=0.023..0.023 rows=31
loops=1)
        Index Cond: (valores2 = 123)

prueba4=# explain analyze select * from vista2;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on tablita  (cost=4.91..63.58 rows=81 width=12) (actual time=0.019..0.043 rows=31
loops=1)
  Recheck Cond: (valores2 = 123)
  Heap Blocks: exact=26
  ->  Bitmap Index Scan on idx1  (cost=0.00..4.89 rows=81 width=0) (actual time=0.013..0.013 rows=31
loops=1)
        Index Cond: (valores2 = 123)
```

Vistas

Desde la versión 9.3 de PostgreSQL las Vistas son “modificables”, es decir se pueden cambiar e insertar los datos en una vista, esta característica viene activa por defecto.

Solo se puede modificar o insertar data en una Vista cuyo query haga una consulta plana a una tabla.

```
pruebas=# create table pruebas (id int, nombre varchar(2));

pruebas=# create view vista1 as select * from pruebas;

pruebas=# insert into vista1 (id, nombre) values (3,'AB');
pruebas=# insert into vista1 (id, nombre) values (4,'BC');

pruebas=# select * from pruebas;
 id | nombre
----+--------
  3 | AB
  4 | BC
(2 rows)
```

Vistas

```
pruebas=# create view vista2 as select nombre from pruebas;
CREATE VIEW

pruebas=# insert into vista2 (nombre) values ('DC');
pruebas=# update vista2 set nombre = 'DA' where nombre = 'AB';

pruebas=# select * from pruebas;
 id | nombre
----+--------
  4 | BC
    | DC
  3 | DA
(3 rows)
```

```
pruebas=# create view vista3 as select a.id, b.nombre from pruebas a join
pruebas b on a.id = b.id;

pruebas=# insert into vista3 (id,nombre) values (10,'ZZ');
ERROR:  cannot insert into view "vista3"
DETAIL:  Views that do not select from a single table or view are not
automatically updatable.
HINT:  To enable inserting into the view, provide an INSTEAD OF INSERT
trigger or an unconditional ON INSERT DO INSTEAD rule.
```

Vistas

Las versiones anteriores a la 9.3 no no permiten modificar los datos de una vista, pero es posible simular esto generando un Rule para la Vista que redireccione cualquier intento de modificación hacia la tabla original.

```
prueba4=# create table tbl_vista ( id integer, nombre varchar(100));

prueba4=# crea view vw_vista as select * from tbl_vista;

prueba4=# create rule rul_vista as on insert to vw_vista do instead insert
into tbl_vista (id,nombre) values (new.id, new.nombre);

prueba4=# insert into vw_vista (id,nombre) values (1,'ernesto');

prueba4=# select * from tbl_vista;
 id | nombre
----+---------
  1 | ernesto
```

Vistas Materializadas

Una vista materializada a diferencia de una vista convencional, es en realidad un tabla se crea a partir de un query, esta tabla puede ser indexada, su gran limitación es que la actualización de la data debe ser realizada a mano.

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] table_name
    [ (column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]
```

```
prueba=# create table numeros ( id serial, campo int);
Insertamos miles de datos

prueba=# create materialized view vw_mat_numeros as
                 select * from numeros where campo=10;
SELECT 65536      <-- creamos la vista materializada con datos

prueba=# create materialized view vw_mat_numeros2 as
                 select * from numeros where campo=10 with no data;
SELECT 0          <-- creamos la vista materializada sin datos
```

Vistas Materializadas

Toda Vista Materializada necesita ser refrescada para actulizar los datos que almacena.

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] name
    [ WITH [ NO ] DATA ]
```

```
prueba=# select * from vw_mat_numeros2;
ERROR:  materialized view "vw_mat_numeros2" has not been populated
HINT:  Use the REFRESH MATERIALIZED VIEW command.
```

Actualizamos la data en la vista:

```
prueba=# refresh materialized view vw_mat_numeros2;
REFRESH MATERIALIZED VIEW
```

```
prueba=# select count(*) from vw_mat_numeros2;
 count
-------
 65536
```

Vistas Materializadas

Creamos una vista y cambiamos los nombres a los campos

```
prueba=# create materialized view vw_mat_numeros3 (Tid,Tcampo) as
              select * from numeros where campo=20;
SELECT 65536

prueba=# select * from vw_mat_numeros3 limit 3;
 tid | tcampo
-----+--------
   2 |     20
   7 |     20
  12 |     20

prueba=# create index idx_vista on vw_mat_numeros3 (Tid);
CREATE INDEX     <-- creamos un índice en la vista materializada
```

Vistas Materializadas

```
prueba=# explain analyze select * from vw_mat_numeros3 where tid > 100
and tid < 500 limit 10;
                                QUERY PLAN
-----------------------------------------------------------------------
Limit  (cost=0.29..1.54 rows=10 width=8) (actual time=0.018..0.025
rows=10 loops=1)
   ->  Index Scan using idx_vista on vw_mat_numeros3  (cost=0.29..9.81
rows=76 width=8) (actual time=0.016..0.020 rows=10 loops=1)
         Index Cond: ((tid > 100) AND (tid < 500))
 Total runtime: 0.115 ms
(4 rows)
```

Para refrescar una vista podemos usar el parámetro CONCURRENTLY para que al hacer el refresh no afecte las lecturas que otros pudieran estar realizando en esa vista.

Transacciones

El modelo ACID

A <-- Atomicidad: es la propiedad que asegura que la operación se ha realizado o no, y por lo tanto ante un fallo del sistema no puede quedar a medias.
C <-- Consistencia: es la propiedad que asegura que sólo se empieza aquello que se puede acabar. Por lo tanto se ejecutan aquellas operaciones que no van a romper la reglas y directrices de integridad de la base de datos.
I <-- Aislamiento (ISOLATION): es la propiedad que asegura que una operación no puede afectar a otras. Esto asegura que la realización de dos transacciones sobre la misma información sean independientes y no generen ningún tipo de error.
D <-- Durabilidad: es la propiedad que asegura que una vez realizada la operación, ésta persistirá y no se podrá deshacer aunque falle el sistema.

PostgreSQL cumple todas estas condiciones.

Transacciones

La estructura básica de una transacción es:

```
BEGIN
  --- Querys ---
COMMIT ó si hay problemas ROLLBACK

prueba=# create table tbl_transaccion ( id integer);

prueba=# begin;
prueba=# insert into tbl_transaccion values(1);
prueba=# insert into tbl_transaccion values(2);
prueba=# commit;
prueba=# select * from tbl_transaccion;
 id
----
  1
  2
```

Transacciones

```
prueba=# begin;
prueba=# insert into tbl_transaccion values(3);
prueba=# insert into tbl_transaccion values(4);
prueba=# rollback;
prueba=# select * from tbl_transaccion;
 id
----
  1
  2
```

Transacciones

SAVEPOINT, nos permite determinar en que punto deseamos ir grabando la información que vamos procesando antes de ejecutar un rollback en caso de problemas.

```
prueba5=# select * from tbl_usuario;
 id |   nombre
----+------------
  8 | lucho
  9 | pedro
  2 | juan
  3 | pedro

prueba5=# begin;
prueba5=# insert into tbl_usuario (nombre) values ('manuel');
prueba5=# insert into tbl_usuario (nombre) values ('rosa');
prueba5=# savepoint sv_uno;
prueba5=# insert into tbl_usuario (nombre) values ('carmen');
prueba5=# rollback to savepoint sv_uno;
prueba5=# insert into tbl_usuario (nombre) values ('lucia');
prueba5=# commit;
```

Transacciones

```
prueba5=# select * from tbl_usuario;
 id |   nombre
----+-------------
  8 | lucho
  9 | pedro
  2 | juan
  3 | pedro
 10 | manuel
 11 | rosa
 13 | lucia
```

Transacciones

El comando LOCK nos permite bloquear una tabla.

```
prueba=# begin;
BEGIN
prueba=# lock tbl_transaccion in exclusive mode;
LOCK TABLE
```

En otra consola intentar esto:

```
prueba=# update tbl_transaccion set id = id * 2;    <-- no culmina se
queda en espera
```

En la consola anterior hacer “commit” y el update culminará.

Sin embargo si hacemos un “select” a esa tabla en otra consola la información procesará sin ningún problema, los niveles de bloqueo permiten ejecutar cierto tipo de operaciones.

Transacciones

Tipos de LOCK:

ACCESS SHARE
ROW SHARE
ROW EXCLUSIVE
SHARE UPDATE EXCLUSIVE
SHARE
SHARE ROW EXCLUSIVE
EXCLUSIVE
ACCESS EXCLUSIVE

https://www.postgresql.org/docs/10/static/explicit-locking.html

Transacciones

Hay que ser cuidadoso con este comando, todas las operaciones a una DB ejecutan algún tipo de lock pre-definido y entre ellos pueden llegar a colisionar

Para ver los locks:

```
prueba=# select * from pg_locks;
  locktype  | database | relation | page | tuple | virtualxid | transactionid | classid | objid | objsubid | virtualtransaction |  pid  |      mode       |
granted
------------+----------+----------+------+-------+------------+---------------+---------+-------+----------+--------------------+-------+-----------------
+---------
 relation   |    22073 |    10969 |      |       |            |               |         |       |          | 2/6239             | 17247 | AccessShareLock | t
 relation   |    22073 |    22183 |      |       |            |               |         |       |          | 1/439              |  2498 | ExclusiveLock   | t
 virtualxid |          |          |      |       | 1/439      |               |         |       |          | 1/439              |  2498 | ExclusiveLock   | t
 virtualxid |          |          |      |       | 2/6239     |               |         |       |          | 2/6239             | 17247 | ExclusiveLock   | t
```

Transacciones

Para ver los querys que están ejecutando los locks:

```
SELECT
    waiting.locktype           AS waiting_locktype,
    waiting.relation::regclass AS waiting_table,
    waiting_stm.current_query  AS waiting_query,
    waiting.mode               AS waiting_mode,
    waiting.pid                AS waiting_pid,
    other.locktype             AS other_locktype,
    other.relation::regclass   AS other_table,
    other_stm.current_query    AS other_query,
    other.mode                 AS other_mode,
    other.pid                  AS other_pid,
    other.granted              AS other_granted
FROM
    pg_catalog.pg_locks AS waiting
JOIN pg_catalog.pg_stat_activity AS waiting_stm  ON ( waiting_stm.procpid = waiting.pid  )
JOIN pg_catalog.pg_locks AS other ON ((waiting."database" = other."database" AND waiting.relation  =
other.relation)
        OR waiting.transactionid = other.transactionid )
JOIN pg_catalog.pg_stat_activity AS other_stm ON (other_stm.procpid = other.pid)
WHERE NOT waiting.granted AND waiting.pid <> other.pid
```

```
 waiting_locktype |  waiting_table  |             waiting_query             |   waiting_mode   | waiting_pid | other_locktype |
other_table   |      other_query      |    other_mode    | other_pid | other_granted
-----------------+-----------------+---------------------------------------+------------------+-------------+----------------
+-----------------+----------------------+-----------------+-----------+--------------
 relation         | tbl_transaccion | update tbl_transaccion set id = id * 4; | RowExclusiveLock |       17741 | relation       |
tbl_transaccion | <IDLE> in transaction | RowExclusiveLock |      2498 | t
 relation         | tbl_transaccion | update tbl_transaccion set id = id * 4; | RowExclusiveLock |       17741 | relation       |
tbl_transaccion | <IDLE> in transaction | ExclusiveLock    |      2498 | t
```

Roles y Accesos

Un rol es básicamente lo mismo que un usuario ó un grupo de usuarios, no existe diferencia fundamental entre ambos, solo al crear un usuario tiene más permisos asignados que cuando uno crea un rol donde todo esta desactivado por defecto.

```
CREATE ROLE name [ [ WITH ] option [ ... ] ]
where option can be:
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE rolename [, ...]
    | IN GROUP rolename [, ...]
    | ROLE rolename [, ...]
    | ADMIN rolename [, ...]
    | USER rolename [, ...]
    | SYSID uid
```

Roles y Accesos

```
prueba=# create role rol1;
prueba=# create user user1;
prueba=# \du
                             List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication | {}
 rol1      | Cannot login                                               | {}
 user1     |                                                            | {}
```

Roles y Accesos

Los permisos.

Por defecto uno tiene permisos totales sobre los objetos que haya creado con su propio rol, ningún rol puede acceder a un objeto creado por otro rol.

Se usan 2 comandos principales para asignación de permisos Grant (autoriza) y Revoke (desautoriza).

```
prueba4=# create role usr1 password 'prueba';
prueba4=# create role usr2 password 'prueba';

Como usr1
prueba4=> create table tbl_prueba ( id integer);

Como usr2
prueba4=> select * from tbl_prueba;
ERROR:  permission denied for relation tbl_prueba
```

Roles y Accesos

GRANT

- **GRANT ON TABLE**
- **GRANT (column) ON TABLE**
- **GRANT ON SEQUENCE**
- **GRANT ON DATABASE**
- **GRANT ON FOREIGN DATA WRAPPER <-- para conexiones Dlink**
- **GRANT ON FOREIGN SERVER <-- para conexiones Dlink**
- **GRANT ON FUNCTION**
- **GRANT ON LANGUAGE**
- **GRANT ON LARGE OBJECT**
- **GRANT ON SCHEMA**
- **GRANT ON TABLESPACE**
- **GRANT role_name TO role_name**

Roles y Accesos

- **GRANT ON TABLE**
- **GRANT (column) ON TABLE**

```
prueba4=# create role usr3 password 'usr3' login;
prueba4=# create table tbl_acceso (id integer, nombre varchar(100));
prueba4=# grant insert on table tbl_accesos to usr3;

Como usr3

prueba4=> select * from tbl_acceso;
ERROR:  permission denied for relation tbl_acceso

prueba4=> insert into tbl_acceso values ( 1, 'ernesto');
INSERT 0 1
```

Roles y Accesos

GRANT ON TABLE
- **GRANT (column) ON TABLE**

```
prueba4=# grant select (nombre) on tbl_acceso to usr3;
GRANT
```

Como usr3

```
prueba4=> select * from tbl_acceso;
ERROR:  permission denied for relation tbl_acceso

prueba4=> select nombre from tbl_acceso;
 nombre
---------
 ernesto
```

No existe GRANT para “vistas”, se tratan como tablas (usar ON TABLE).

Roles y Accesos

GRANT ON DATABASE
- **GRANT ON SCHEMA**

```
prueba4=# create schema sin_acceso;
prueba4=# create table sin_acceso.prueba(id integer);
CREATE TABLE
```

Como usuario usr3

```
prueba4=> select * from sin_acceso.prueba;
ERROR:  permission denied for schema sin_acceso
LINE 1: select * from sin_acceso.prueba;
```

Como superusuario:

```
prueba4=# grant usage on sin_acceso to usr3;
prueba4=# grant select on table sin_acceso.prueba to usr3;
```

Roles y Accesos

REVOKE

Sigue más o menos la misma estructura de GRANT salvo que en vez de dar permisos los retira.

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES
          TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

```
prueba4=# revoke all on sin_acceso.prueba from ur3;
```

Roles y Accesos

Usuarios del dbms vs Usuarios de la aplicación

Muchas aplicaciones (especialmente web) siguen la lógica de conectarse a la dbms usando un solo usuario y controlan los accesos por opciones del sistema.
Esto es muy práctico de mantener, no llena la dbms de usuarios, PERO, permite que todos tengan el mismo nivel de acceso a todos los objetos de la db con los problemas que esto puede tener.
Para ver los accesos superiores de un rol puede utilizar este comando:

```
prueba4=> select * from pg_roles;
  usename   | usesysid | usecreatedb | usesuper | usecatupd |  passwd  | valuntil | useconfig
------------+----------+-------------+----------+-----------+----------+----------+-----------
 postgres   |       10 | t           | t        | t         | ******** |          |
 dbadmin    |    16392 | t           | t        | t         | ******** |          |
 pgsql      |    16643 | t           | t        | t         | ******** |          |
 usr3       |    24410 | f           | f        | f         | ******** | infinity |
 usr4       |    24414 | f           | f        | f         | ******** |          |
```

```
prueba4=> select * from pg_user; <-- lista solo aquellos ROLEs creados como USERs.
```

Roles y Accesos

Ojo, todas las claves por defecto se guardan encriptadas en MD5 (lo cual es relativamente seguro, pero no totalmente) sin embargo cerciórese de no generar claves sin encriptación ya que fácilmente se puede ver estas con el siguiente comando:

```
template1=# create role usr8 unencrypted password 'se_me_ve_todo'
login;

template1=# select usename, passwd from pg_shadow;
  usename  |               passwd
-----------+-------------------------------------
 postgres  | md54abfba9c735cfbc34d97b56a593120c0
 dbadmin   | md5695c0e25f6fe7c4ce633c67292190b90
 pgsql     |
 usr1      | md51575823e236277b188fdd5d691aa8d08
 usr3      | md5f06099b5e97add7ed510d76e24146f1e
 usr8      | se_me_ve_todo
```

Roles y Accesos - Grupos

Un grupo básicamente se crea a partir de un ROL que hereda sus permisos a todos aquellos que han realizado un GRANT hacia el.

PASO 1: Como usuario administrador hacemos lo siguiente:

```
prueba=# create user grupo;
prueba=# create user user1;
prueba=# alter user grupo password 'grupo';
prueba=# alter user user1 password 'user1';

prueba=# \du
                             List of roles
 Role name |                   Attributes                   | Member of
-----------+------------------------------------------------+-----------
 grupo     |                                                | {}
 postgres  | Superuser, Create role, Create DB, Replication | {}
 User1     |                                                | {}

prueba=# create table tbl2 (campo int);
CREATE TABLE
```

Roles y Accesos - Grupos

PASO 2: Como usuario “grupo” hacemos lo siguiente:

```
prueba=> create table tbl1 (campo int);
CREATE TABLE
```

La regla de accesos indica que uno solo puede ver los objetos que uno mismo crea.

PASO 3: Como usuario “user1” hacemos lo siguiente:

```
prueba=> select * from tbl1;
ERROR:  permission denied for relation tbl1
```

La regla se cumple, “user1” no puede ver lo que el usuario “grupo” ha creado, ahora vamos a darle accesos heredados.

PASO 4: como usuario administrador hacemos lo siguiente:

```
prueba=# grant grupo to user1;
GRANT ROLE
prueba=# \du
                             List of roles
 Role name |                   Attributes                   | Member of
-----------+------------------------------------------------+-----------
 grupo     |                                                | {}
 postgres  | Superuser, Create role, Create DB, Replication | {}
 user1     |                                                | {grupo}
```

Roles y Accesos - Grupos

PASO 5: Como usuario “user1” intentamos ingresar a la tabla “tbl1”:

```
prueba=> select * from tbl1;
 campo
-------
(0 rows)
```

En el ejemplo “user1” ahora esta configurado para heredar cada privilegio que el usuario “grupo” tenga configurado, esto se actualiza de manera automática, apenas se cambie el usuario “grupo” todos sus herederos se actualizan, podemos verlo en el siguiente ejemplo:

PASO 6: Como usuario “grupo” intentamos ingresar a “tbl2” que ha sido creado por el usuario “postgres”.

```
prueba=> select * from tbl2;
ERROR:  permission denied for relation tbl2
```

Como se ve no puede ingresar porque esa tabla la creo el usuario “postgres”, esto mismo sucederá si es que “user1” desea ingresar a esa tabla.

PASO 7: como usuario administrador damos accesos al usuario “grupo”.

```
prueba=# grant select on tbl2 to grupo;
GRANT
```

Roles y Accesos - Grupos

PASO 8: Como usuario “grupo” ahora intentamos ingresar nuevamente a la tabla “tbl2”:

```
prueba=> select * from tbl2;
 campo
-------
(0 rows)
```

PASO 9: Como usuario “user1” intentamos lo mismo.

```
prueba=> select * from tbl2;
 campo
-------
(0 rows)
```

Como puede verse en el ejemplo apenas se actualizó los permisos de “grupo” todos sus herederos obtuvieron los permisos que el ahora tiene.

Luego que un usuario hereda los accesos de un un grupo, cualquier eevocación de permisos que se le intente hacer, sobre aquello que su grupo tenga permisos, no tendrá efecto.

Roles y Accesos – Ver los GRANT

Grant por tabla:

```
prueba=# \z tbl1
                            Access privileges
 Schema | Name | Type  |  Access privileges  | Column access privileges
--------+------+-------+---------------------+--------------------------
 public | tbl1 | table | grupo=arwdDxt/grupo |
```

Grant por Usuario Y Tablas:

```
prueba=# SELECT * FROM information_schema.role_table_grants;
```

```
grantor  | grantee  |table_catalog  |table_schema |table_name| privilege_type | is_grantable | with_hierarchy
---------+--------+---------------+-------------+----------+----------------+--------------+---------------
 postgres | postgres| prueba        | public      | tbl2     | REFERENCES     | YES          | NO
 postgres | postgres| prueba        | public      | tbl2     | TRIGGER        | YES          | NO
 postgres | grupo   | prueba        | public      | tbl2     | SELECT         | NO           | YES
```

Roles y Accesos – Ver los GRANT

Otras tablas con información de ROLEs y GRANTs:

```
prueba=# SELECT table_schema, table_name
           FROM information_schema.tables
           where table_name like '%role%';

    table_schema    |            table_name
--------------------+----------------------------------
 pg_catalog         | pg_roles
 information_schema | applicable_roles
 pg_catalog         | pg_db_role_setting
 information_schema | administrable_role_authorizations
 information_schema | enabled_roles
 information_schema | role_column_grants
 information_schema | role_routine_grants
 information_schema | role_table_grants
 information_schema | role_udt_grants
 information_schema | role_usage_grants
```

ADMINISTRACION BASICA

Configuración - Archivo pg_hba.conf

“pg_hba.conf” nos permite definir (en orden de aplicación):

a)Conecciones locales (x socket) ó de host seguras o inseguras (x TCP/IP)
b)Base datos a la que nos podemos conectar
c)Usuarios que se pueden conectar a las DBs
d)Direcciones autorizadas para conectarse (solo en caso TCP/IP)
e)Tipo de autenticación

```
# Database administrative login by UNIX sockets
local   all         postgres                            ident
# TYPE  DATABASE    USER        CIDR-ADDRESS            METHOD
# "local" is for Unix domain socket connections only
local   all         all                                 ident
# IPv4 local connections:
host    all         all         127.0.0.1/32            ident
# IPv6 local connections:
host    all         all         ::1/128                 md5
```

Configuración - Archivo pg_hba.conf

a)Conecciones locales (x socket) ó de host seguras o inseguras (x TCP/IP)

Posibles valores:

```
Local      <-- conexiones por socket, más rápidas, solo si la
               aplicación que hará uso de la db esta en el mismo
               servidor
Host       <-- lo normal, conexiones vendrán por la red, puede o
               no tener Soporte SSL
Hostssl    <-- solo conexión SSL
Hostnossl  <-- solo sin SSL
```

Configuración - Archivo pg_hba.conf

b)Base datos a la que nos podemos conectar

Se pueden definir conexiones para todas las db usando “ALL” o una específica mencionándola directamente.

Recomendación: no usen combinaciones de mayúsculas y minúsculas.

Se puede usar sameuser, samegroup, samerol para indicar que la base de datos a la que me conecto es la misma que mi nombre de usuario, grupo ó rol.

c)Usuarios que se pueden conectar a las Dbs

Podemos especificar restricciones para todos los usuarios usando “ALL” o para uno específico escribiendo su nombre, si pones “+” por detrás es el nombre de un grupo.

En b) y c) podemos poner varias db y usuarios separados por comas.

Configuración - Archivo pg_hba.conf

d)Direcciones autorizadas para conectarse (solo en caso TCP/IP)

Podemos especificar IPs directamente o por rango

```
192.168.1.10/32  <-- solo la IP mencionada
192.168.1.0/24   <-- todo el segmento 192.168.1.x
192.168.0.0/16   <-- todo el segmento 192.168.x.x
192.0.0.0/8      <-- todo el segmento 192.x.x.x
0.0.0.0/0        <-- todo el mundo
```

También podemos especificar IP y netmask (poco usado).

Se puede especificar IP de diferentes redes (no todo debe estar en 192.x.x.x).

Esto solamente se utiliza en el escenario donde el primer parámetro es HOST, HOSTSSL ó HOSTNOSSL.

Configuración - Archivo pg_hba.conf

e)Tipo de autenticación

PostgreSQL provee diferentes tipos de autenticación, los mas comunes:

```
Trust    <-- permite que se pueda ingresar a la DB sin password
Reject   <-- cierra el acceso a una DB o a las direcciones especificadas
Password <-- requiere que se especifique el password del usuario, el
             password viaja en texto plano
MD5      <-- requiere un password, pero este va en formato MD5
Ident    <-- usa el usuarios del OS para ingresar a la db, usada en
             conexiones locales
Ldap     <-- para autentificaciones contra un servicio como OpenLdap
Peer     <-- fuerza a que el usuario del OS sea el mismo de la DB, solo
             se puede usar en conexiones locales en sistemas Linux/Unix.
```

gss, sspi, krb5, radius, cert, pam son de menos uso y requieren configuraciones de otros servicios (como Ldap).

Cualquier cambio en este archivo requiere reiniciar el servicio del PostgreSQL.

Configuración - Archivo postgresql.conf

El archivo “postgresql.conf” contiene los parámetros de configuración del rendimiento y funcionalidad del dbms, algunos parámetros principales:

```
listen_addresses = 'localhost' <-- permite especificar si las conexiones a
                                   la db son locales o de red (poner “*”).
Port = 5432                    <-- es el puerto TCP que va a escuchar el
                                   Dbms para aceptar conexiones, es
                                   posible tener diferentes instalaciones
                                   en el mismo servidor pero especificando
                                   diferentes puertos.
max_connections = 100          <-- máximo de conexiones simultaneas que
                                   aceptará el servidor.
shared_buffers = 24MB          <-- indica la cantidad de RAM que usa el
                                   Dbms para trabajar las consultas, no
                                   puede ser mayor a el valor guardado en
                                   /proc/sys/kernel/shmmax
                                   Modificar /etc/sysctl.conf añadiendo
                                   kernel.shmmax=[nuevo_valor_en_#entero]
```

Configuración - Archivo postgresql.conf

El archivo “postgresql.conf” contiene los parametros de configuración del rendimiento y funcionalidad del dbms, algunos parámetros principales:

autovacuum = on **<-- activa la “limpieza” de páginas no-activas en el directorio donde se almacena la data, tomar en cuenta que este es un “lazy vacuum” no un “full vacuum” y que los valores de precisión del vacuum no deben generar congestión en la base de datos.**

lc_messages = 'es_PE.utf8' **<-- determina el tipo de “locale” para el almacenamiento de la data (internacionalización), por defecto se usa el mismo del sistema operativo, recomendación no modificarlo.**

Cualquier cambio en este archivo requiere reiniciar el servicio del PostgreSQL.

Logs bajo Gnu/Linux

Los logs se configuran en el archivo “postgresql.conf”, por defecto el directorio de localización de los Logs en Ubuntu Server es “/var/log/postgresql”, o puede estar en “/var/lib/postgresql/9.3/main”.

```
log_destination = 'stderr' <-- permite generar el log de la dbms.
                               Stderr, manda el log a donde se
                               especifique Syslog, manda el log a
                               /var/log/messages

logging_collector = off    <-- permite redirigir el log a archivos
                               alternos

log_directory = 'pg_log'   <-- se puede configurar diversos
                               directorios de Log, este es el por
                               defecto, si no se especifica ruta se
                               asume la de “main” por defecto.

log_connections = off      <-- permite loggear las conexiones que
                               recibe la Dbms, ayuda mucho a depurar
                               el pg_hba.conf

log_disconnections = off   <-- Loggear conexiones cerradas.
```

Logs bajo Gnu/Linux

```
log_directory = 'pg_log'   <-- directorio pajo /var/lib/pgsql/data
                               donde se almacenarán los archivos de
                               log, el Usuario “postgres” debe tener
                               derechos de escritura Sobre el nuevo
                               directorio.

log_filename  = 'postgresql-%a.log' <-- nombre del archivo de log.
                                        dbms, ayuda mucho a depurar el
                                        pg_hba.conf

log_truncate_on_rotation = on  <-- permite generar un nuevo archivo de
                                   log cuando llega a un tamaño
                                   determinado o tiene una antigüedad
                                   determinada.

log_rotation_age = 1d      <-- antigüedad máxima

log_rotation_size = 0      <-- tamaño máximo en bytes que puede tener
                               el archivo de log.

log_duration = off         <-- permite loggear la duración de la
                               ejecución de los querys.
```

Logs bajo Gnu/Linux

```
log_min_duration_statement = -1     <-- permite loggear la duración de la
                                        ejecución de los querys. (valor 0 para
                                        activar, mayor a 0 solo loggerá el tiempo
                                        de los querys que duren más del tiempo
                                        indicado en milisegundos)

log_statement = 'none'      <-- permite loggear los querys que procesa la
                                Dbms.
                                none, no loggea querys
                                all, loggea todos los querys, bien o mal
                                     Procesados (ideal para ambiente de
                                     desarrollo)
                                ddl, loggea los comandos que modifican
                                     Estructuras en la db
                                mod, loggea los mismos comandos que ddl más
                                     Comandos de movimiento de data

client_min_message = notice      <-- tipo de mensaje enviado al cliente
```

Logs bajo Gnu/Linux

```
log_min_messages = warning      <-- tipo de mensaje ha ser escrito en el
                                    log

log_min_error_statement = error <-- identifica a partir de que nivel de
                                    mensaje se debe enviar un mensaje de
                                    error.
```

OjO: estos parámetros añaden tiempo de escritura a disco y consumen un importante espacio.