Autorización en PostgreSQL

Documento de apoyo para cursos

Charles Clavadetscher

31.08.2015

There is no such thing as perfect security, only varying levels of insecurity.

—Salman Rushdie

Tabla de materias

1 Introducción

PostgreSQL ofrece un poderoso sistema de roles para controlar la forma en que los usuarios pueden acceder a los objetos en la base de datos. En el contexto de este artículo, "objetos" son tablas, vistas, secuencias y funciones, es decir las partes de una base de datos con las que interactúa un usuario normal, incluyendo usuarios técnicos compartidos por aplicaciones. Como veremos, antes de poder acceder a ese tipo de objetos, los usuarios necesitan tener privilegios sobre los contenedores en los que se encuentran, en particular bases de datos y esquemas.

En este artículo, se asume, por supuesto que el administrador de la base de datos (DBA) tiene acceso superuser a un cluster PostgreSQL. El término "superuser" se refiere a un tipo de usuario muy particular que no está sujeto a los mecanismos de control de acceso. Es decir que tiene todos los derechos sobre todos los objetos en el clúster. A menudo este usuario se llama "postgres", siendo este el nombre estándar durante la instalación. La instalación de PostgreSQL en sí queda fuera de los límites de este artículo.

Hablando sobre acceso a sistemas es imprescindible diferenciar entre autenticación y autorización. Autenticación se refiere al proceso con el cual un usuario se identifica hacia un sistema. Es decir que la cuestión es si el usuario que está intentando acceder el sistema es realmente la persona (o en algunos casos la aplicación) quien afirma ser. Entre los mecanismos habituales para implementar una autenticación los más conocidos son el uso de un nombre de usuario y una palabra clave o de un certificado digital. La autenticación es un proceso que siempre antecede la autorización.

Con autorización se entiende el conjunto de actividades que un usuario puede cumplir en un sistema. Otro término habitual y a menudo utilizado como sinónimo de autorización es lista de control de acceso (o su correspondiente en inglés access control list, ACL). En el caso de una base de datos esto resulta ser, por ejemplo, el derecho o privilegio de leer o modificar datos de una tabla.

Los ejemplos de código son producidos utilizando psql. psql es el cliente que es instalado por defecto con el software del clúster PostgreSQL. Es posible, claro está, utilizar otros clientes, como por ejemplo pgAdminIII o DbVisualizer. Dado que va a ser importante en este contexto diferenciar los usuarios que ejecutan comandos, la línea de comando tiene el formato:

```
username@database[=|-][#|>]
```

Donde “=” indica la primera línea de un comando y “-” las siguientes. Los signos “#” y “>” indican respectivamente si el usuario con el nombre username es un superuser o no. Por ejemplo

```
postgres@admin=#
admin@uci=>
```

quiere decir que el usuario “postgres” está conectado con la base de datos “admin” como superuser y que el usuario “admin” está conectado con la base de datos uci como usuario normal. Cabe mencionar que usuarios pueden tener atributos que le otorguen algunas capacidades especiales, sin ser superuser (p. ej. crear bases de datos o usuarios). Para formatizar la línea de comando de esta manera, entre en psql los comandos reproducidos aquí abajo.

```
\set PROMPT1 '%n@%/=%# '
\set PROMPT2 '%n@%/-%# '
```

2 Conceptos básicos y definiciones

PostgreSQL es un gestor de bases de datos relacionales organizado en diferentes niveles. El nivel más externo es el clúster. En un servidor se pueden instalar varios clústeres del gestor, siempre y cuando cada uno de ellos utilice un puerto diferente. El puerto por defecto es 5432 y es el mismo que asumiremos en este artículo. Un clúster contiene a su vez bases de datos. Una base de datos (database) es una colección de esquemas (schema), quienes a su vez, finalmente contienen los objetos que se conocen habitualmente en el contexto de técnicas de almacenamiento de datos, tales como tablas (table), vistas (view), secuencias (sequence) y así seguido. Es posible isolar cada usuario utilizando una base de datos por cada usuario. Lo mismo se puede alcanzar utilizando un esquema por cada usuario dentro de una sola base de datos. Cuáles de estas soluciones es la más apropiada depende de los requerimientos que se imponen al diseño del sistema. Sistemas que tienen que compartir datos optarán por poner estos datos en uno o más esquemas compartidos por los usuarios mientras que aplicaciones totalmente independientes podrían estar isoladas en sus propias bases de datos. Una base de datos recién creada en PostgreSQL siempre tiene un esquema `public`[1]. Los privilegios por defecto del esquema `public` permiten a todos los usuarios de crear objetos en él. Lo que a primera vista parece ser, o puede volverse en un problema de seguridad, puede tener ciertas ventajas, como se verá más adelante.

En PostgreSQL los usuario existen a nivel de clúster. Es decir que un usuario no está vinculado a una base de datos específica por defecto. El término utilizado en PostgreSQL para usuarios es rol (`ROLE`). Los roles no son tan solo usuarios sino que pueden ser grupos o ambos. En versiones anteriores a la 8.1, usuarios y grupos eran entidades distintas. Un usuario es un rol con el atributo (`LOGIN`), es decir con el derecho de conectarse a una base de datos. Cabe mencionar que con este atributo no se especifica para cuáles bases de datos el atributo es válido. Una manera de especificar la base de datos es utilizando el fichero de configuración `pg_hba.conf` (vease para detalles [RN11] y [Pos15a], Cap. 19.1). En la instalación por defecto la configuración permite el acceso a todos y para todas las bases de datos desde la misma compudadora en donde se encuentra el clúster (`localhost`). Este hecho no tiene por qué ser negativo. Como veremos más adelante es posible controlar el acceso a

[1] De hecho nuevas bases de datos son creadas como copia de la base de datos estándar "template1". Es decir que si un administrador borró el esquema public de esta base de datos, todas las nuevas bases de datos serán creadas sin este esquema.

una base de datos con los mecanismos de autorización. Estos últimos son particularmente útiles en el caso en que un clúster es gestionado físicamente por una entidad externa (p.ej. el departamento de informática de una empresa) y el responsable de las bases de datos sólo tiene acceso al gestor mas no al sistema operativo[2].

Autorización en PostgreSQL es, entonces, la posibilidad de otorgar privilegios sobre objetos en un clúster o una base de datos a uno o más usuarios o grupos con el objetivo de minimizar el impacto producido por errores de programación en aplicaciones o por usuarios con objetivos dañinos[3].

Antes de entrar en los detalles de la autorización en PostgreSQL cabe mencionar el aspecto de la propriedad sobre objetos. Todos los objetos en un clúster, incluyendo bases de datos, tienen un proprietario. Por diseño el proprietario de un objeto es el usuario que creó el objeto (aunque es posible modificar el proprietario posteriormente) y tiene todos los privilegios que existen sobre este objeto. Usuarios que no son proprietarios de un objeto no tienen ningún privilegio sobre estos, de no ser que el proprietario o un superuser les haya otorgado alguno, ya sea directa o indirectamente por medio de un grupo o de `PUBLIC`.

[2]El centro de investigaciones coyunturales del instituto politécnico federal de Zurich (ETHZ), p.ej. tiene un pequeño grupo de especialistas en informática para gestionar entre otros los datos de las encuestas. El gestor está instalado en un servidor Linux al que este grupo no tiene acceso. El grupo tiene sin embargo acceso al gestor por el puerto 5432 y tiene cuando menos un superuser.

[3]Por lo general una empresa tiene confianza en sus empleados. Existen sin embargo cantidades de ejemplos de actividades criminales cometidas precisamente por empleados o por personas ajenas, cuyo acceso es obtenido de usuarios legítimos por medio de ingeniería social. Básicamente los seres humanos suelen ser el anillo más débil de la cadena de seguridad de un sistema informático.

3 Acceso a la base de datos

El acceso a un clúster de PostgreSQL siempre exige la definición de una base de datos. Vimos en la sección antecedente que un clúster puede tener varias bases de datos. Apenas instalado un clúster tiene tres bases de datos para uso interno del gestor, esto implica que la primera tarea de cada DBA va a ser la creación de una base de datos. Dado que al comienzo no existe todavía ninguna base de datos definida para el uso de usuarios, podemos conectar a la base de datos estándar `postgres`. Si Usted no tiene todavía un superuser para la gestión del clúster, tendrá que conectarse como el superuser `postgres`. Es recomendable crear un superuser diferente de `postgres` para simplificar la gestión[1] y un usuario con privilegios `CREATEDB` y `CREATEROLE` para la gestión de las bases de datos. Con este último podemos crear una base de datos que nos servirá para ilustrar con ejemplos los conceptos descritos.

```
$ sudo -u postgres psql
postgres@postgres=# CREATE ROLE admin CREATEDB CREATEROLE LOGIN PASSWORD 'xxx';
postgres@postgres=# \c - admin
admin@postgres=> CREATE DATABASE uci;
admin@postgres=> \c uci
admin@uci=>
```

La siguiente tarea consisten en crear unos usuarios que puedan utilizar la base de datos `uci`. La creación de usuarios en PostgreSQL requiere un superuser o un usuario con el atributo `CREATEROLE`. El comando SQL para crear un rol es estándar y su forma general puede averiguarse en la documentación oficial de PostgreSQL [Pos15a] o en un terminal psql con el comando `\h CREATE ROLE`.

```
admin@uci=> CREATE ROLE user1 LOGIN PASSWORD 'xxx';
admin@uci=> CREATE ROLE user2 LOGIN PASSWORD 'xxx';
admin@uci=> \du
                             List of roles
     Role name     |       Attributes         |         Member of
-------------------+--------------------------+---------------------------
 admin             | Create role, Create DB   | {}
 user1             |                          | {}
 user2             |                          | {}
```

[1] Esa es una medida de comodidad. Si Usted prefiere puede conectarse siempre como postgres

Hasta este punto no hemos tomado ninguna medida para modificar la configuración por defecto de la base de datos. A nivel de base de datos esto quiere decir que todos los usuarios existentes en el clúster pueden conectarse a `uci`. Es posible ver los privilegios configurados para una base de datos con el comando `\l` (lista de las bases de datos).

```
§
admin@uci=> \l
                                      List of databases
     Name      |   Owner   | Encoding |   Collate   |    Ctype    |  Access privileges
---------------+-----------+----------+-------------+-------------+---------------------
 uci           | admin     | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
```

El campo con los privilegios está vacío lo que significa que los valores por defecto están en uso. Por lo general los privilegios por defecto se refieren a `PUBLIC` o sea privilegios otorgados a todos los usuarios en el clúster. El encontrar cuáles existen efectivamente puede ser una tarea algo difícil como leer toda la documentación o buscar informaciones en internet. Una manera más eficiente es utilizar las funciones presentes en cada instalación de PostgreSQL para este fin ([Pos15a] Tab. 9-57, p. 284). La forma más general de estas funciones es[2]:

```
has_<object_type>_privilege(user, object_name, privilege);
```

Todas estas funciones retornan un valor boolean (t, f) indicando si el usuario tiene o no el privilegio indicado sobre el objeto. Para averiguar si todos los usuarios pueden conectarse a la base de datos, podemos entonces ejecutar la función como sigue.

```
admin@uci=> SELECT has_database_privilege('public', 'uci', 'connect');
 has_database_privilege
------------------------
 t
```

En un clúster con varias bases de datos podemos limitar el acceso quitándole el privilegio a `PUBLIC` y otorgándoselo a nuestros usuarios.

```
admin@uci=> REVOKE ALL ON DATABASE uci FROM PUBLIC;
admin@uci=> GRANT CONNECT ON DATABASE uci TO user1;
```

Si averiguamos ahora quiénes pueden conectar con la misma función de antemano observamos lo siguiente.

[2]Según el objeto investigado pueden haber más parámetros. El conjunto de funciones existe también sin el parámetro user. En estos casos user es implicitamente el usuario que esta conectado en la base datos y que ejecuta la función.

```
admin@uci=> SELECT has_database_privilege('public', 'uci', 'connect') AS "public",
                   has_database_privilege('user1', 'uci', 'connect') AS "user1",
                   has_database_privilege('user2', 'uci', 'connect') AS "user2";
 public | user1 | user2
--------+-------+-------
 f      | t     | f
```

PUBLIC no puede conectarse y por ende tampoco user2 al que no le hemos otorgado el privilegio. Ahora es obvio que si un usuario no puede conectarse a una base de datos, tampoco puede ocasionar daños. Así tan solo con esta medida ya mejoramos la seguridad de nuestra base de datos considerablemente[3].

Además de CONNECT el nivel de base de datos conoce dos privilegios más:

TEMPORARY: Permite al usuario crear tablas temporales. Estas tablas sólo existen durante la sesión y desaparecen una vez que el usuario se desconecta de la base de datos.

CREATE: Permite al usuario crear esquemas en la base de datos. Es importante no confundir el significado de CREATE en diferentes niveles. Como veremos más adelante un privilegio con el mismo nombre existe a otros niveles, pero con otro significado.

[3]De hecho un usuario tiene la posibilidad de otorgar su rol a otros roles, otorgandole con lo mismo todos sus privilegios, incluido el de conectarse a todas las bases de datos, a las que el usuario tiene acceso. Cabe notar que este comportamiento coincide con la documentación y no parece ser considerado un error por la comunidad. En la sección sobre la organización del accesso volveremos sobre este punto.

4 Acceso a esquemas

Un esquema se puede entender como una sección dentro de una base de datos. Puesto que todos los demás objetos de una base de datos están contenidos en un esquema es imprescindible que haya al menos uno. Cuando se crea una base de datos, esa tiene por defecto un esquema public. Como el nombre dice el esquema es creado de tal manera que todos los usuarios pueden crear otros objetos en ello. Los privilegios que existen a nivel de esquema diferencian entre la posibilidad de crear nuevos objetos (`CREATE`) y utilizar objetos existentes (`USAGE`). Para ver quién tiene cuál privilegio sobre un esquema se pueden utilizar las funciones que mencionamos en la sección antecedente o el comando `\dn+`.

```
admin@uci=> \dn+
                          List of schemas
  Name  |  Owner   |  Access privileges   |      Description
--------+----------+----------------------+------------------------
 public | postgres | postgres=UC/postgres+| standard public schema
        |          | =UC/postgres         |
```

La lista de acceso nos dice que los usuarios postgres y `PUBLIC` tienen privilegios tanto `USAGE` como `CREATE` sobre el esquema public. La segunda línea no menciona un usuario antes de "=" lo que es equivalente a `PUBLIC`. Después del signo "/" en cada línea vemos el usuario quien ha otorgado el privilegio - en este caso postgres. Además vemos que el propietario es también postgres. La implicación de esto es que nuestro usuario admin no tienen suficientes privilegios para modificar los privilegios de acceso al esquema. Un superuser o el propietario del esquema podrían hacerlo.

Por lo general la existencia de un esquema con estas características no es necesariamente un problema. De hecho hasta hay circunstancias en donde un esquema public tiene ventajas, por lo menos en cuanto se refiere al privilegio `USAGE`. Por ejemplo, funciones, extensiones, operadores o hasta vistas compartidas por varios usuarios podrían implementarse en este esquema. Lo que sí vale la pena limitar es la creación de nuevos objetos. De esta manera podemos crear un espacio controlado en el que solo unos cuantos pueden escribir y todos pueden leer. El primer paso para alcanzar este objetivo es hacer las modificaciones del acceso como acabamos de mencionar por medio de un superuser. La forma general de los comandos para otorgar y revocar privilegios se muestra aquí seguido.

```
GRANT <list of privileges>
ON [<object type] <list of object names>
TO <list or roles>;

REVOKE <list of privileges>
ON [<object type] <list of object names>
FROM <list or roles>;
```

La lista de privilegios disponibles depende del objeto indicado en object type y los elementos tienen que estar separados por comas. La mención de object type se puede omitir si el objeto es una tabla o una vista. Los elementos de lista de objetos también tienen que estar separados por comas, así como los de la lista de roles. Es lógico que los objetos en la lista tienen que ser del mismo tipo. Existen además opciones para otorgar o revocar privilegios sobre todos los objetos de un mismo tipo en un esquema. Es posible permitirle a un usuario al que se le haya otorgado un privilegio de otorgar a su vez estos privilegios a otros usuarios usando la opción `WITH GRANT OPTION` inexistente por defecto. Esta característica puede ser útil en algunas ocasiones, mas por lo general no es necesaria si el DBA quiere mantener el control de acceso centralizado y más estricto. Todas las opciones para los comandos `GRANT` y `REVOKE` están documentadas en [Pos15a], GRANT, pp. 1522-1528. La sinopsis del comando se puede obtener también en psql con el comando `\h [GRANT|REVOKE]`[1]. Con estos conocimientos podemos ahora modificar el acceso al esquema public con los siguientes comandos.

```
postgres@uci=# REVOKE CREATE ON SCHEMA public FROM PUBLIC;
postgres@uci=# GRANT USAGE, CREATE ON SCHEMA public TO admin;
postgres@uci=# \dn+
                          List of schemas
  Name  |  Owner   |  Access privileges   |      Description
--------+----------+----------------------+------------------------
 public | postgres | postgres=UC/postgres+| standard public schema
        |          | =U/postgres         +|
        |          | admin=UC/postgres    |
```

Después vamos a crear un nuevo esquema en donde nuestros usuarios van a poder trabajar. Llamaremos al esquema uci y vamos a otorgar a user1 y user2 todos los privilegios. El usuario admin tiene el privilegio `CREATE` sobre la base de datos, así que puede crear esquemas.

```
admin@uci=> CREATE SCHEMA uci;
admin@uci=> GRANT ALL ON SCHEMA uci TO user1, user2;
```

[1]Con el comando `\h <command>` pueden obtenerse sinopsis para todos los comandos existentes.

```
admin@uci=> \dn+ uci
                 List of schemas
 Name | Owner | Access privileges | Description
------+-------+-------------------+-------------
 uci  | admin | admin=UC/admin   +|
      |       | user1=UC/admin   +|
      |       | user2=UC/admin    |
```

En lugar de otorgar los privilegios a cada uno de los usuarios, hubieramos podido otorgarselos a `PUBLIC` de tal manera que también nuevos usuarios todavía no presentes en el clúster, tuvieran el privilegio por defecto[2].

Antes de seguir con nuestras explicaciones tenemos que abrir un paréntesis sobre el tema visibilidad. Todos los objetos en una base de datos se encuentran en un esquema. Por esto se pueden identificar con una calificación completa según la convención:

```
[<database>.]<schema>.<object_name>[object_parameters]
```

El nombre de la base de datos por lo general no hace falta, porque una conexión siempre implica que el usuario se encuentre en una base de datos específica. El nombre del esquema es indispensable. Para simplificar la gestión de los objetos cada usuario tiene en su perfil una variable llamada search_path. Cuando se arranca una consulta de cualquier tipo, el gestor intentará buscar un objeto, cuyo nombre es igual al que se encuentra en la consulta. Si el nombre está calificado con el esquema, el gestor buscará en el esquema especificado. Si, en cambio, el nombre del objeto no tiene calificación, el gestor buscará el nombre del objeto en la lista ordenada de esquemas definida en search_path. La búsqueda termina cuando el gestor encuentra la primera correspondencia. Si no encuentra ningún objeto con el nombre en la consulta, el gestor arroja un error "does not exist". En el caso de acciones DDL[3] (`CREATE [TABLE|VIEW|...]`) el objeto es creado en el primer esquema de la lista presente en la base de datos. De esta manera es posible poner en search_path todos los esquemas a los que un usuario tiene acceso, aunque estos se encuentren en diferentes bases de datos. Esquemas en la lista que no existen en una base de datos son ignorados. Por defecto search_path tiene el contenido siguiente.

[2]Otra manera más elegante, utilizando grupos, se presentará más adelante en la sección sobre la organización del acceso.

[3]Data Definition Language es el conjunto de comando SQL necesarios para crear, modificar o borrar la estructura de objetos en una base de datos.

```
admin@uci=> SHOW search_path;
  search_path
----------------
 "$user",public
```

El primer valor es una variable para identificar un esquema con el mismo nombre del rol. El esquema public está presente por defecto, porque el esquema mismo es creado en el momento en que se crea la base de datos. El valor de search_path puede ser modificado en la sesión con el comando `SET`. A menudo es preferible modificar el valor de manera durable. En este caso se tiene que modificar el rol mismo. Cada usuario tiene el derecho de modificar este valor para su propio rol. Como de costumbre superusers y usuarios con el atributo `CREATEROLE` pueden modificar todos los roles. Mostramos aquí solo el caso de admin, pero para los ejemplos que siguen hizimos la misma modificación para user1 y user2.

```
admin@uci=> ALTER ROLE admin SET search_path=uci,public;
```

Nótese que para que el cambio tenga efecto es necesario reconectarse a la base de datos. El comando `\c` sin parámetros implica una reconexión con el mismo usuario en la misma base de datos.

```
admin@uci=> \c
You are now connected to database "uci" as user "admin".
admin@uci=> SHOW search_path;
      search_path
-----------------------
 uci, public
```

Pusimos el nuevo esquema uci al comienzo de la lista para que nuevos objetos sin calificación sean creados allí. Si queremos permitir que un usuario cree objetos en un esquema sin calificarlos, es importante que el nombre del esquema se encuentre antes de cualquier otro que existe en la base de datos sobre el que el usuario no tiene el privilegio `CREATE`. En el siguiente ejemplo el usuario user1 construye la lista en search_path sin considerar sus privilegios sobre los esquemas. Anteriormente definimos que user1 no puede crear objetos en public, pero sí en uci. Si public se encuentra antes de uci en search_path y user1 intenta crear una tabla sin especificar el esquema, PostgreSQL intentará crear la tabla en public y arrojará un error porque el usuario no tiene el privilegio necesario en public. Si la tabla está calificada por completo o si el esquema uci se encuentra antes de public, user1 puede crear el objeto.

```
You are now connected to database "uci" as user "user1".
user1@uci=> SET search_path=public,uci;
user1@uci=> CREATE TABLE test_user1_1 (id INTEGER);
ERROR:  permission denied for schema public
user1@uci=> CREATE TABLE uci.test_user1_1 (id INTEGER);
user1@uci=> SET search_path=uci,public;
user1@uci=> CREATE TABLE test_user1_2 (id INTEGER);
user1@uci=> \dt test_user1_*
           List of relations
 Schema |     Name     | Type  | Owner
--------+--------------+-------+-------
 uci    | test_user1_1 | table | user1
 uci    | test_user1_2 | table | user1
```

Finalmente es importante entender que aunque un usuario tenga privilegios sobre una tabla si el esquema en el que esta se encuentra no está en la lista de search_path, la tabla parecerá invisible (siempre y cuando no se califique con el esquema).

```
user1@uci=> SET search_path=public;
user1@uci=> SELECT * FROM test_user1_1;
ERROR:  relation "test_user1_1" does not exist
user1@uci=> RESET search_path;
user1@uci=> SELECT * FROM test_user1_1;
 id
----
(0 rows)
```

El comando `RESET` pone el valor configurado en el rol al comienzo de la sesión actual. Esto implica que aunque se cambie el valor de search_path con el comando `ALTER ROLE` esto no va a ser reconocido por `RESET`[4].

Es posible ver el contenido de search_path para todos los usuarios utilizando una consulta hacia la tabla pg_roles. Esta consulta puede ser hecha para todos los usuarios[5].

```
admin@uci=> SELECT rolname, rolconfig FROM pg_roles
admin@uci-> WHERE rolname LIKE 'user%' OR rolname = 'admin';
 rolname |          rolconfig
---------+-----------------------------
 admin   | {"search_path=uci, public"}
 user1   | {"search_path=uci, public"}
 user2   | {"search_path=uci, public"}
```

El otorgar el privilegio `CREATE` sobre un esquema es razonable en situaciones en el que

[4]Dejamos como ejercicio para el lector averiguar la veracidad de esta declaración.

[5]El acceso para leer tablas del catálogo es otorgado por defecto a todos los usuarios. No existe aún claridad hasta que punto esto pueda ser un problema para la seguridad del sistema.

los usuarios mismos son responsables de la estructura de sus datos. Hoy en día con herramientas como SCRUM hay tendencias en integrar el desarrollo de la base de datos de una aplicación en procesos iterativos, así que los requerimientos se van integrando poco a poco. En estos casos el responsable de la base de datos, si es miembro del equipo de desarrollo, toma parte activa en el diseño de las estructuras necesarias y puede integrar su concepto de control de acceso desde un comienzo[6]. Si esto no es posible, probablemente y a más tardar en el momento en que la base de datos tendrá que integrarse en un proceso de producción, se tendrán que averiguar y revisar las estructuras e imponer los niveles de control de acceso que sean necesarios. También hay casos en los que el crear objetos en un esquema es dictado por la tecnología usada para el desarrollo de la aplicación. En el KOF p. ej. las aplicaciones web son implementadas usando el framework Django, que requiere este tipo de privilegios para crear y manipular tablas para el control de acceso a nivel de aplicación. Lo que hicimos en el KOF fue de crear un esquema dedicado y con el nombre "django" para el framework, cuyo usuario tiene el mismo nombre y es el proprietario del esquema. Los datos de la empresa en cambio, se encuentran en otros esquemas con un control de acceso mucho más estricto y adecuado a las necesidades de seguridad de la empresa. Esto demuestra que distintos requerimientos pueden coexistir en la misma base de datos.

Hasta este punto hemos mostrado que es posible dejarle mucha libertad a los usuarios, sea esto el resultado de fuerza mayor o de libre arbitrio del responsable de la base de datos. Para los siguientes capítulos y para mejor ilustración de los conceptos, vamos a revocar el privilegio `CREATE` de user1 y user2 sobre el esquema uci. Todos los objetos serán creado entonces por el usuario admin, quien también definirá el control de acceso sobre ellos.

[6]Esta forma de trabajar puede poner ciertos problemas en la gestión del acceso si el proceso de desarrollo requiere la creación de nuevos objeto. Nuevos objetos sólo permiten el acceso del proprietario y privilegios para otros tienen que otorgarse específicamente. En la sección sobre Default Privileges veremos una posibilidad para solucionar este tipo de problemas de antemano.

5 Acceso a otros objetos de una base de datos

Para visualizar los comentarios que siguen vamos a crear una tabla en el esquema uci[1].

```
CREATE TABLE uci.catalogue (
  item_id SERIAL,
  name TEXT NOT NULL,
  location TEXT NOT NULL DEFAULT 'Main store',
  price NUMERIC(10,2) NOT NULL DEFAULT 0.00,
  responsible TEXT NOT NULL DEFAULT session_user,
  PRIMARY KEY (item_id)
);
charles@uci=# \dp
                                  Access privileges
 Schema |         Name          |   Type   | Access privileges | Column access privileges
--------+-----------------------+----------+-------------------+--------------------------
 uci    | catalogue             | table    |                   |
 uci    | catalogue_item_id_seq | sequence |                   |
```

Como se ve en el ejemplo con el comando `\dp` se pueden ver los privilegios presente sobre tablas, vistas y secuencias en la base de datos.

5.1 Tablas, vistas y secuencias

Cuando se oye la palabra "base de datos" lo primero que se nos ocurre es por lo general una tabla. Eso no es raro puesto que una tabla es un contenedor para el capital no humano más precioso de una empresa. Tablas son, pues, el fundamento de las actividades económicas, sean estas informaciones para la gestión de los clientes, de los productos y almacenes, para el monitoreo de compras y ventas o la creación de estadísticas para decidir sobre estrategias de mercado por mencionar unos pocos ejemplos. Es claro que si los datos de la empresa tienen tanto valor, es necesario tomar medidas de protección para evitar que estos se pierdan, sean robados o manipulados. En las secciones anteriores vimos como controlar el acceso hasta los esquemas. Aquí entramos ahora en los detalles sobre los privilegios que existen sobre tablas y vistas. Una vista es una pieza de código SQL que arroja lo que

[1] El enfoque de este artículo es el uso de los privilegios de acceso. La tabla en uso para los ejemplo pudiera obviamente normalizarse, pero este no es el objetivo que aquí perseguimos.

desde el punto de vista del usuario parece ser una tabla. De hecho la tabla retornada por una vista puede ser el resultado de complejas operaciones sobre varias tablas, incluyendo modificaciones o valores derivados de los datos para la presentación.

Por defecto el usuario que crea una tabla o una vista es el proprietario y el único que tiene cualquier tipo de acceso. Otros usuarios necesitan recibir privilegios para poder utilizar la tabla. El comando `GRANT` permite utilizar la palabra clave `ALL` para otorgar todos los privilegios sobre una tabla. Nótese que todos los privilegios no implica que otro usuario pueda destruir la tabla (`DROP TABLE`). Sin embargo hay que tener algo de cuidado, pues otorgar todos los privilegios es por lo general muy fácil pero casi nunca es lo que se quiere obtener.

Las secuencias son objetos que retornan un número que se puede utilizar p. ej. como primary key de una tabla en el caso en que no existan valores simples o combinados que por su lógica permitan identificar de manera unívoca a una fila en una tabla. El uso de secuencias se hace por medio de funciones, cuyo acceso no es otorgado por defecto al crearse la secuencia. Según los privilegios otorgados a un usuario sobre una tabla que utiliza una secuencia, puede ser que sea necesario añadir privilegios sobra dicha secuencia.

Vamos a empezar por ver los privilegios que más se conocen y que habitualmente son los únicos que tienen sentido.

`SELECT` es el privilegio que se necesita para leer filas de una tabla. El privilegio se puede otorgar para todas las columnas de una tabla o solo para un subconjunto. Nótese que este privilegio también permite el uso del comando `COPY TO` con el que se pueden exportar datos para un sistema externo. Para una secuencia el efecto de `SELECT` es el permiso de utilizar la función currval que retorna el valor actual de la secuencia. En general este derecho por sí solo no tiene mucho sentido ya que el valor está en uso en alguna otra parte.

Para añadir filas se necesita el privilegio `INSERT`. Como para `SELECT` se puede permitir insertar valores para todas las columnas o solo algunas. En este caso el proporcionar la lista de las columnas que se pueden insertar implica limitaciones en el uso del comando SQL `INSERT`. Los campos que quedan indefinidos en el comando `INSERT` tienen que tener valores por defecto. De forma análoga al anterior privilegio, este también permite el uso de `COPY FROM` que se puede utilizar para importar datos. Si la tabla en la se quieren añadir datos tiene una o más secuencias será necesario otorgar el permiso de ejecutar la función nextval. Eso se alcanza otorgando el privilegio `USAGE` sobre la secuencia. En el ejemplo que sigue más abajo se muestra este mecanismo en la práctica.

El privilegio `UPDATE` permite hacer cambios en una o más filas de la tabla. Es posible proporcionar una lista de las columnas que el usuario puede modificar. Antes de poder hacer las modificaciones el gestor tiene que encontrar las filas afectadas, lo que requiere el privilegio `SELECT`. Para secuencias `UPDATE` permite utilizar las funciones nextval y setval.

La segunda función es una de esas funciones que en operaciones normales no es necesario utilizar y sirve para poner un valor específico en la secuencia. Usuarios normales no tienen por lo general por qué utilizarla[2].

En las operaciones de una empresa suele suceder que algunas filas por una u otra razón se volvieron obsoletas y pueden borrarse de la tabla. Para este fin se puede otorgar el privilegio `DELETE`. Así como para `UPDATE`, el gestor tiene que buscar antes las filas que tiene que borrar. Por eso de hecho usuarios que reciben el privilegio `DELETE` deben recibir también el privilegio `SELECT`.

Vamos ahora a poner en práctica lo que acabamos de describir. Para empezar vamos a otorgarles todos los cuatro privilegios a user1 y a insertar una fila en la tabla. Para estos ejemplos vamos a pasar constantemente del usuario admin al usuario user1 y viceversa. Fíjese bien en cada línea del código, cual usuario está intentando ejecutar cuál comando.

```
admin@uci=> GRANT SELECT, INSERT, UPDATE, DELETE ON uci.catalogue TO user1;

user1@uci=> INSERT INTO catalogue (name, price)
user1@uci-> VALUES ('PostgreSQL Security',12.40);
ERROR:  permission denied for sequence catalogue_item_id_seq
```

En la parte teórica vimos que para insertar filas en una tabla que tiene una secuencia es indispensable otorgar el privilegio sobre la secuencia al usuario. El nombre de la secuencia se ve en el mensaje del gestor. También es posible ver el nombre de la secuencia visualizando la estructura de la table con el comando `\d <table name>`

```
user1@uci=> \d catalogue
                                  Table "uci.catalogue"
   Column    |     Type      |                          Modifiers
-------------+---------------+-------------------------------------------------------------
 item_id     | integer       | not null default nextval('catalogue_item_id_seq'::regclass)
 name        | text          | not null
 location    | text          | not null default 'Main store'::text
 price       | numeric(10,2) | not null default 0.00
 responsible | text          | not null default "session_user"()
Indexes:
    "catalogue_pkey" PRIMARY KEY, btree (item_id)
```

Entonces vamos a otorgar el privilegio que falta e intentar insertar la fila de nuevo.

[2]Secuencias por si bien que son muy útiles deberían de funcionar de manera transparente, es decir sin acciones por parte del usuario. Una búsqueda en internet sobre el tema "PostgreSQL problems with sequences" retorna suficiente material de aclaramiento.

```
admin@uci=> GRANT USAGE ON SEQUENCE catalogue_item_id_seq TO user1;

user1@uci=> INSERT INTO catalogue (name, price)
user1@uci-> VALUES ('PostgreSQL Security',12.40);
user1@uci=> SELECT * FROM catalogue;
 item_id |        name         |  location  | price | responsible
---------+---------------------+------------+-------+-------------
       1 | PostgreSQL Security | Main store | 12.40 | user1
```

Lo siguiente que queremos alcanzar es modificar los privilegios para evitar que el usuario pueda insertar valores para item_id y para responsable. La razón para lo primero es que siendo una secuencia no quisiéramos que el usuario tome control de estos valores. Si un usuario entra un valor para item_id (acción perfectamente legal) la función nextval de la secuencia no es invocada. El resultado es que en el momento en que algún usuario intente insertar una fila utilizando el valor por defecto de item_id (de la secuencia) recibirá un error de violación de primary key.

```
user1@uci=> INSERT INTO catalogue (item_id,name,price)
user1@uci-> VALUES (2,'Troubleshooting PostgreSQL',12.20);
admin@uci=> SELECT * FROM catalogue;
 item_id |            name            |  location  | price | responsible
---------+----------------------------+------------+-------+-------------
       1 | PostgreSQL Security        | Main store | 10.50 | user1
       2 | Troubleshooting PostgreSQL | Main store | 12.20 | user1


user1@uci=> INSERT INTO catalogue (name,price)
user1@uci-> VALUES ('PostgreSQL From The Trenches',15.00);
ERROR:  duplicate key value violates unique constraint "catalogue_pkey"
DETAIL:  Key (item_id)=(2) already exists.
```

En estos casos no queda más remedio que poner el valor de la secuencia manualmente.

```
admin@uci=> SELECT setval('catalogue_item_id_seq'::regclass,2);
user1@uci=> INSERT INTO catalogue (name,price)
user1@uci-> VALUES ('PostgreSQL From The Trenches',15.00);
user1@uci=> SELECT * FROM catalogue;
 item_id |             name             |  location  | price | responsible
---------+------------------------------+------------+-------+-------------
       1 | PostgreSQL Security          | Main store | 10.50 | user1
       2 | Troubleshooting PostgreSQL   | Main store | 12.20 | user1
       3 | PostgreSQL From The Trenches | Main store | 15.00 | user1
```

En el caso del campo "responsible" queremos evitar que un empleado delegue su responsabilidad sobre la fila a otro sin pasar, p. ej., por un sistema de control (p. ej. el jefe).

El usuario en responsible es además el mismo que creó la fila (valor por defecto).

Como vimos en la parte teórica es posible limitar tanto `INSERT` como `UPDATE` a ciertas columnas. Al comienzo de la parte práctica hemos otorgado a user1 ambos privilegios para todas las columnas. La primera idea sería pues revocar el privilegio sobre las columnas como aquí se muestran.

```
admin@uci=> REVOKE INSERT (item_id, responsible),
admin@uci-> UPDATE (item_id, responsible) ON catalogue FROM user1;
admin@uci=> \dp
                                  Access privileges
 Schema |         Name          |   Type   |  Access privileges  | Column access privileges
--------+-----------------------+----------+---------------------+-------------------------
 uci    | catalogue             | table    | admin=arwdDxt/admin+|
        |                       |          | user1=arwd/admin    |
 uci    | catalogue_item_id_seq | sequence | admin=rwU/admin    +|
        |                       |          | user1=U/admin       |
```

Desgraciadamente esto no lleva al resultado esperado y además el gestor no da ningún mensaje que deje imaginar que el comando no haya sido ejecutado. La forma correcta para llegar al objetivo es revocar primero los privilegios sobre toda la tabla para luego otorgar los privilegios solo para las columnas que se quieren.

```
admin@uci=> REVOKE INSERT, UPDATE ON catalogue FROM user1;
admin@uci=> GRANT INSERT (name,location,price),
admin@uci-> UPDATE (name,location,price) ON catalogue TO user1;
```

De esta manera aparecen ahora los privilegios sobre las columnas como queríamos lograr.

```
admin@uci=> \dp
                                  Access privileges
 Schema |         Name          |   Type   |  Access privileges  | Column access privileges
--------+-----------------------+----------+---------------------+-------------------------
 uci    | catalogue             | table    | admin=arwdDxt/admin+| name:                  +
        |                       |          | user1=rd/admin      |   user1=aw/admin       +
        |                       |          |                     | location:              +
        |                       |          |                     |   user1=aw/admin       +
        |                       |          |                     | price:                 +
        |                       |          |                     |   user1=aw/admin
 uci    | catalogue_item_id_seq | sequence | admin=rwU/admin    +|
        |                       |          | user1=U/admin       |
```

Volvemos en la base de datos como user1 para averiguar los efectos de esta configuración.

```
user1@uci=> INSERT INTO catalogue (item_id,name,price,responsible)
user1@uci-> VALUES (10,'PostgreSQL Performance',15.00,'user2');
ERROR:  permission denied for relation catalogue


user1@uci=> INSERT INTO catalogue (name,price)
user1@uci-> VALUES ('PostgreSQL Performance',15.00);
user1@uci=> UPDATE catalogue SET price = 11.30, responsible = 'user2' WHERE item_id = 1;
ERROR:  permission denied for relation catalogue
user1@uci=> UPDATE catalogue SET price = 11.30 WHERE item_id = 1;
user1@uci=> SELECT * FROM catalogue;
 item_id |             name             |  location  | price | responsible
---------+------------------------------+------------+-------+-------------
       1 | PostgreSQL Security          | Main store | 11.30 | user1
       2 | Troubleshooting PostgreSQL   | Main store | 12.20 | user1
       3 | PostgreSQL From The Trenches | Main store | 15.00 | user1
       4 | PostgreSQL Performance       | Main store | 15.00 | user1
```

Así como esperábamos, user1 puede ejecutar solo operaciones que no comprometan el contenido de las columnas item_id y responsible. Nótese que el mensaje de error claramente define que su origen se encuentra en el sistema de autorización de la base de datos. La visión de conjunto librada por el comando `\dp` en este caso permite un análisis muchos más sencillo que el uso de las funciones que vimos en la sección "Acceso a la base de datos", ya que cada campo debería averiguarse singularmente.

```
admin@uci=> SELECT has_table_privilege('user1','catalogue','insert') AS table_insert,
admin@uci-> has_column_privilege('user1','catalogue','price','insert')
admin@uci-> AS column_price_insert;
 table_insert | column_price_insert
--------------+---------------------
 f            | t
```

Utilizando privilegios hasta el nivel de columna permite por lo tanto una enorme flexibilidad y granularidad. De la misma manera como se muestra arriba, podríamos definir los privilegios para otros usuarios. Por ejemplo el jefe de user1 pudiera tener el privilegio de modificar el contenido de la columna responsible con el fin de delegar la gestión de una o más filas a otro empleado. Es claro que la configuración a este nivel de detalle puede volverse muy compleja.

Además de los privilegios que acabamos de ver existen unos cuantos más a nivel de tabla. Nos limitamos aquí a una descripción sumaria, porque en la mayoría de los casos no va ser necesario otorgarlos a usuarios normales.

`TRUNCATE`: Con este privilegio y el correspondiente comando del mismo nombre se puede vacíar toda la tabla de una vez. Usuarios con el privilegio `DELETE` tienen también la posi-

bilidad de vaciar la tabla usando p. ej. `DELETE FROM <table name>`. La diferencia está en el manejo de transacciones. `TRUNCATE` es mucho más eficiente ya que al llegar un `COMMIT` el gestor puede simplemente reemplazar la tabla con los nuevos datos, si es que hay algunos. En cambio `DELETE` marcaría cada fila como "dirty" y el espacio de almacienamento quedaría ocupado hasta el próximo `VACUUM`. El segundo proceso necesita mucho más tiempo, sobre todo si tenemos una tabla con muchas filas. Cuando una tabla se utiliza en un entorno para test o entrenamiento puese ser razonable otorgar este privilegio a un usuario, dado que a menudo hay que reinicializar la base de datos vaciando todas las entradas hecha durante los tests o el entrenamiento. En un entorno productivo es muy importante averiguar en detalle, si un usuario necesita realmente este privilegio.

`REFERENCES`: Este tipo de privilegio se puede otorgar para todas las columnas de una tabla o para una selección y permite crear un `FOREIGN KEY`. Obviamente el usuario quien recibe este privilegio tiene también que recibirlo sobre la tabla referenciada. Puesto que se trata de un privilegio estrictamente relacionado con el desarrollo del diseño de la base de datos, en general usuarios no requieren que se les otorgue[3].

`TRIGGER`: A nivel de tabla el privilegio `TRIGGER` le permite al usuario de asociar una función disparadora con la tabla (`CREATE TRIGGER`). Nótese que esto no tiene nada que ver con los privilegios para crear o ejecutar funciones disparadoras. Estos últimos son el tema de las sección siguiente.

5.2 Funciones

En PostgreSQL funciones son un medio para extender la funcionalidad del gestor. Funciones pueden ser escritas en varios lenguajes y cumplir con tareas que serían imposible o muy difícil de alcanzar usando nada más que el lenguaje estándar SQL. El tema funciones en generales está fuera de los límites de nuestro tema y daremos por supuesto que el lector ya tenga los conocimientos necesarios para entender los ejemplos de código que se mostrarán en el séquito. A parte de las explicaciones en la documentación oficial de Postgresql [Pos15a] y en la publicación de Krosing et al [KMR13], lectores de habla hispana pueden encontrar una excelente guía en la obra de Sotolongo y Vazquez [SLVO15].

En esta sección queremos observar el privilegio `EXECUTE`, el único que se puede otorgar o revocar sobre funciones, y sus efectos. Cabe mencionar que este privilegio no tiene

[3] En la sección sobre el acceso a esquemas mencionamos el caso de un framework que genera tablas como parte de su funcionalidad. En este caso es necesario que el usuario del framework tenga el privilegio `REFERENCES` siempre y cuando el framework utilice esta capacidad de la base de datos. En el caso del KOF el problema es solucionado implicitamente, porque el framework utiliza un esquema dedicado y es su proprietario.

influjo sobre la capacidad de escribir funciones sino tan solo sobre la capacidad de ejecutar una función. Para que un usuario pueda escribir funciones necesita tener el privilegio `USAGE` sobre el lenguaje en el que se quiere escribir la función. Funciones no solo permite encapsular lógica compleja detrás de un interfaz sencillo, sino que pueden usarse para el control de acceso, como veremos en la sección sobre funciones declaradas como "security definer".

El privilegio `EXECUTE` es otorgado por defecto a `PUBLIC` al crear la función. Aunque a primeras parezca atrevido otorgarle este privilegio a todos, existen excelentes razones para ello e inclusive no tiene un impacto directo sobre la seguridad de la base de datos. Siendo extensiones de las capacidades del gestor, funciones deberían ser ejecutables para todos. Un caso muy evidente es el uso de funciones para implementar operadores. La capacidad de ejecutar una función no otorga automáticamente privilegios sobre los objetos manipulados por la misma[4]. Con este punto resulta claro la importancia que tiene el control de acceso sobre los objetos que vimos anteriormente.

Veremos con dos funciones sencillas que hacen lo mismo, o sea devolver un texto, pero se encuentran en diferentes esquemas. Asímismo para nuestro primer ejemplo vamos a revocar el privilegio `USAGE` sobre el esquema uci de user2. Nótese además que usamos el mismo nombre para la función. Esto es posible porque se encuentran en distintos esquemas.

```
admin@uci=> REVOKE USAGE ON SCHEMA uci FROM user2;
```

Por medio de este comando user2 a lo presente no tiene ningún privilegio sobre el esquema uci. Con el usuario admin creamos las funciones como sigue.

```
CREATE FUNCTION uci.show_schema()
RETURNS TEXT
AS $$
BEGIN
  RETURN 'I am in schema uci';
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION public.show_schema()
RETURNS TEXT
AS $$
BEGIN
  RETURN 'I am in schema public';
END;
$$ LANGUAGE plpgsql;
```

[4]En la sección sobre funciones security definer veremos como es posible en casos controlados obviar a este impedimento.

Al invocar las funciones por nuestro usarios user1 y user2 recibimos diferentes respuestas. PostgreSQL busca en los esquemas en search_path hasta encontrar una función que cumpla con la signatura. En el caso de user1, pues la primera función encontrada se encuentra en el esquema uci. En el caso de user2, aunque también tenga el esquema uci en su search_path y el privilegio `EXECUTE`[5] sobre la función no puede llamar la versión en uci porque le revocamos el privilegio `USAGE` sobre el esquema. Si la función con el mismo nombre no existiera en public, user2 recibiría un error, como mostramos en el siguiente snippet.

```
user1@uci=> SELECT * FROM show_schema();
     show_schema
---------------------
 I am in schema uci

user2@uci=> SELECT * FROM show_schema();
      show_schema
-----------------------
 I am in schema public

admin@uci=> DROP FUNCTION public.show_schema();

user2@uci=> SELECT * FROM show_schema();
ERROR:  function show_schema() does not exist
```

En algunas ocasiones, y esta es una de estas, los mensajes que acompañan un error pueden sonar algo misteriosos. Con el comando `\df` se obtiene una lista de las funciones presentes en la base de datos[6]. La función existe y el mensaje de error pretende lo contrario.

```
admin@uci=> \df
                            List of functions
 Schema |    Name     | Result data type | Argument data types |  Type
--------+-------------+------------------+---------------------+--------
 uci    | show_schema | text             |                     | normal
```

Ya vimos en una ocasión precedente que un mensaje del tenor "... does not exist" puede indicar que el esquema falte en el search_path del usuario. Pero sabemos que user2 tiene exactamente el mismo valor que user1. En este caso es necesario averiguar que los privilegios en objetos sobreordenados sean presentes. Para esta ilustración le habíamos revocado a user2 el privilegio de utilizar el esquema uci. Es lógico pues que, si user2 no tiene acceso

[5] Para mayor exactitud el privilegio es otorgado a `PUBLIC` y no directamente a user2.

[6] La lista solo contiene las funciones escritas por los usuarios. Para ver también las funciones del sistema se puede utilizar `\dfS`. También hay que mencionar el hecho de funciones con el mismo nombre solo se muestran una vez y más concretamente la version que se encuentra en el primer esquema de search_path.

al esquema, tampoco puede invocar funciones allí disponibles. Entonces vamos a otorgar otra vez el derecho a user2.

```
admin@uci=> GRANT USAGE ON SCHEMA uci TO user2;
user2@uci=> SELECT * FROM show_schema();
     show_schema
---------------------
 I am in schema uci
```

Si queremos limitar el uso de una función explícitamente a uno o más usuarios tendremos que hacer unas modificaciones en los privilegios.

```
admin@uci=> REVOKE EXECUTE ON FUNCTION uci.show_schema() FROM PUBLIC;
admin@uci=> GRANT EXECUTE ON FUNCTION uci.show_schema() TO user1;

user2@uci=> SELECT * FROM show_schema();
ERROR:  permission denied for function show_schema

user1@uci=> SELECT * FROM show_schema();
     show_schema
---------------------
 I am in schema uci
```

Nótese que el privilegio se tiene que revocar de `PUBLIC`. Es un error muy común revocar el privilegio del usuario que se quiere excluir. Esta técnica no llevaría al resultado esperado.

Como dijimos anteriormente un usuario que tenga el privilegio `EXECUTE` sobre una función, ya sea directa o indirectamente, no recibe automaticamente privilegios sobre otros objetos que aparecen en el cuerpo de la función. Pongamos que tengamos una función que lee datos de la tabla catalogue.

```
CREATE FUNCTION uci.read_catalogue(p_item_id INTEGER)
RETURNS uci.catalogue
AS $$
  SELECT * FROM uci.catalogue
  WHERE item_id = p_item_id;
$$ LANGUAGE sql;
```

Come era de esperarse nuestros usuarios pueden ejecutar la función.

```
admin@uci=> SELECT has_function_privilege('user1','read_catalogue(INTEGER)','execute')
admin@uci-> has_function_privilege('user2','read_catalogue(INTEGER)','execute')
 user1_execute | user2_execute
---------------+---------------
 t             | t
```

Si ambos usuarios invocan la función vemos que solo uno de ellos recibe un resultado.

```
user1@uci=> SELECT * FROM read_catalogue(1);
 item_id |        name         |  location  | price | responsible
---------+---------------------+------------+-------+-------------
       1 | PostgreSQL Security | Main store | 11.30 | user1
(1 row)

user2@uci=> SELECT * FROM read_catalogue(1);
ERROR:  permission denied for relation catalogue
```

El mensaje de error declara justamente que el usuario user2 no tiene los privilegios necesarios sobre la tabla catalogue. La tabla está protegida por medio de los privilegios otorgados sobre ella como volvemos a mostrar aquí seguido.

```
admin@uci=> \dp+ catalogue
                              Access privileges
 Schema |   Name    | Type  |  Access privileges  | Column access privileges
--------+-----------+-------+---------------------+--------------------------
 uci    | catalogue | table | admin=arwdDxt/admin+| name:                   +
        |           |       | user1=rd/admin      |   user1=aw/admin        +
        |           |       |                     | location:               +
        |           |       |                     |   user1=aw/admin        +
        |           |       |                     | price:                  +
        |           |       |                     |   user1=aw/admin
```

Un tipo especial de funciones son las funciones disparadoras que se ejecutan cuando un usuario hace cambios en una tabla (para los detalles sobre este tipo de función véase [SLVO15]). Por su misma naturaleza las funciones disparadoras no presentan problemas desde un punto de vista de la seguridad porque no pueden ser invocadas directamente. Eso solo es posible en el contexto de un trigger. Es claro que para que la función sea invocada el usuario tiene que tener el privilegio necesario sobre la tabla. Un usuario, p. ej. que no tenga el privilegio de insertar datos en una tabla tampoco va a poder invocar la función disparadora. En el contexto de funciones disparadoras es muy fácil olvidar en alguna parte algún privilegio. Vamos a suponer que nuestro admin quiera mantener un mínimo de información sobre las actividades de user1 y user2. Para hacer esto podría crear una función disparadora y crear un trigger sobre la tabla catalogue. Para empezar vamos a crear una tabla para recibir las informaciones sobre las actividades.

```
CREATE TABLE uci.log_changes (
  change_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  change_type TEXT,
  change_table TEXT,
  change_user TEXT NOT NULL DEFAULT SESSION_USER
);
```

El siguiente paso es crear una función disparadora para recolectar los datos a poner en la tabla log_changes.

```
CREATE OR REPLACE FUNCTION uci.log_changes()
RETURNS TRIGGER
AS $$
BEGIN
  INSERT INTO uci.log_changes (change_type, change_table)
  VALUES (TG_OP, TG_TABLE_SCHEMA||'.'||TG_TABLE_NAME);
  IF TG_OP = 'DELETE' THEN RETURN OLD;
  ELSE RETURN NEW;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

Finalmente creamos el trigger, es decir asociamos la función disparadora con la tabla indicando en cuáles situaciones se tiene que ejecutar la función.

```
CREATE TRIGGER log_changes
AFTER INSERT OR UPDATE OR DELETE ON uci.catalogue
FOR EACH ROW EXECUTE PROCEDURE uci.log_changes();
```

Antes de volver a nuestro tema vamos a hacer unos pequeños tests para estar seguros de que el mecanismo de base funciona correctamente.

```
admin@uci=> INSERT INTO catalogue (name, price) VALUES ('Principia Mathematica',56.10);
admin@uci=> SELECT * FROM log_changes ;
        change_date         | change_type |  change_table  | change_user
----------------------------+-------------+----------------+-------------
 2015-06-14 10:43:45.298952 | INSERT      |  uci.catalogue | admin

admin@uci=> DELETE FROM catalogue WHERE item_id = 6;
admin@uci=> SELECT * FROM log_changes ;
        change_date         | change_type |  change_table  | change_user
----------------------------+-------------+----------------+-------------
 2015-06-14 10:43:45.298952 | INSERT      |  uci.catalogue | admin
 2015-06-14 10:44:25.571587 | DELETE      |  uci.catalogue | admin
```

Nuestro objetivo es que cuando un usuario haga una operación de manipulación de

datos sobre la tabla se genere una fila en la tabla log_changes.

```
user1@uci=> INSERT INTO catalogue (name, price) VALUES ('Principia Mathematica',56.10);
ERROR:  permission denied for relation log_changes

admin@uci=> GRANT INSERT ON log_changes TO user1;

user1@uci=> INSERT INTO catalogue (name, price) VALUES ('Principia Mathematica',56.10);

admin@uci=> SELECT * FROM log_changes;
        change_date         | change_type |  change_table  | change_user
----------------------------+-------------+----------------+-------------
 [...]
 2015-06-14 10:48:15.934824 | INSERT      | uci.catalogue  | user1
```

Como se ve el primer intento no dio el resultado esperado. El usuario user1 no tiene el privilegio para insertar filas en la tabla log_changes. Por eso nuestro usuario admin (el proprietario de la tabla) tiene que otorgar el privilegio. Una vez hecho todo anda como queríamos. A la misma vez sin embargo, hemos creado una situación que le permite a user1 manipular las informaciones sobre actividades sin usar la función disparadora.

Por ejemplo, con un simple comando SQL user1 puede hacerle creer al sistema que otro usuario borró una fila hace un año. Nuestro log tiene muy poca información, ya que no tenemos como objetivo elaborar sobre técnicas de gestión de datos. En un entorno productivo esta posibilidad potencialmente abre la puerta para abusos. Afortunadamente existen métodos y técnicas para controlar de forma más estricta cómo acceder a tablas.

```
user1@uci=> INSERT INTO log_changes (change_date, change_type, change_table, change_user)
user1@uci-> VALUES (CURRENT_DATE - 360, 'DELETE', 'uci.catalogue', 'user2');

admin@uci=> SELECT * FROM log_changes;
        change_date         | change_type |  change_table  | change_user
----------------------------+-------------+----------------+-------------
 [...]
 2014-06-19 00:00:00        | DELETE      | uci.catalogue  | user2
```

Para terminar esta sección vamos a presentar como se pueden averiguar los privilegios sobre funciones. A diferencia de otros comandos de psql `\df` no retorna ninguna información sobre el control de acceso, tampoco en su forma ampliada `\df+`. Para encontrar la información se puede utilizar la función que vimos al comienzo de esta sección: `has_function_privilege()`. Es también posible consultar el catálogo. La siguiente consulta muestra por ejemplo como obtener la lista de control de acceso de la función show_schema.

```
admin@uci=> SELECT proname, proacl FROM pg_proc WHERE proname = 'show_schema';
   proname   |            proacl
-------------+-------------------------------
 show_schema | {admin=X/admin,user1=X/admin}
```

Este ejemplo es una consulta mínima. Por supuesto es posible perfeccionar la consulta de tal manera que mejor corresponda a las necesidades del DBA. Para esos fines consúltese la descripción de las tablas del catálogo en la documentación oficial de PostgreSQL ([Pos15a]).

Otra posibilidad, quizás algo más cómoda es de consultar el "information_schema" como aquí se muestra.

```
admin@uci=> SELECT grantor, grantee, routine_name, privilege_type
admin@uci-> FROM information_schema.routine_privileges
admin@uci-> WHERE specific_schema = 'uci';
 grantor | grantee | routine_name | privilege_type
---------+---------+--------------+----------------
 admin   | admin   | show_schema  | EXECUTE
 admin   | user1   | show_schema  | EXECUTE
 admin   | PUBLIC  | log_changes  | EXECUTE
 admin   | admin   | log_changes  | EXECUTE
```

5.3 Otros objetos

Los objetos analizados hasta este punto son los que habitualmente son importantes en las tareas de a diario de un DBA. A parte de ellos existen varios objetos más con correspondientes privilegios. Ese nivel de detalle ya está fuera de los límites de este artículo y nos limitaremos por lo tanto para la completitud a mencionar cuáles son, junto con una descripción mínima de lo que son y cuáles privilegios tienen. Los detalles se pueden encontrar en la documentación oficial de PostgreSQL ([Pos15a]).

`LANGUAGE`: El único privilegio que existe es `USAGE`, que permite al usuario escribir funciones en el lenguaje especificado. Es la manera más efectiva de evitar que cualquiera escriba funciones de forma incontrolada. Sin embargo la creación de funciones, como todos los demás objetos, require que el usuario tenga el privilegio `CREATE` sobre el esquema en el que quiere escribir las funciones. Si esto ya está limitado por configuración no hace falta revocar este privilegio de `PUBLIC`. Al instalar un nuevo lenguaje el privilegio `USAGE` es otorgado a `PUBLIC` por defecto. Sin embargo, a pesar de eso, los lenguajes considerados de desconfianza solo puede ser usados por superuser.

`TYPE`: El privilegio `USAGE` permite que se utilice el tipo de datos en la creación de tablas, funciones y otros objetos del esquema. No tiene nada que ver con la creación de nuevos

tipos de datos. Este último es un asunto complejo, ya que crear un nuevo tipo de datos requiere en algunos casos que el usuario sea superuser. En otros (p. ej. tipos de datos compuestos) es suficiente que el usuario tenga el privilegio `USAGE` sobre todos los tipos de datos que figuran en el nuevo tipo de datos. En general no es necesario crear un tipo de datos nuevo, porque los que ya existen cubren la mayoría de las las necesidades. Si se vuelve necesario por cuestiones de diseño crear un tipo de dato, hay que tomar en cuenta que una vez creado, este no tiene todavía los elementos necesarios para optimizar su uso. Concretamente sería eventualmente necesario crear funciones y operadores para que el tipo de datos se pueda indexar.

`DOMAIN`: Las características de `DOMAIN` son básicamente las mismas que las de `TYPE`. Aquí también sólo existe el privilegio `USAGE` que permite su uso en la creación de tablas y otros objetos de un esquema y que no influye sobre la creación misma de un `DOMAIN`. La diferencia sustancial en comparación con `TYPE` es que `DOMAIN` no es realmente un tipo de datos nuevo sino, más bien, un subconjunto de un tipo de datos ya existente. El uso de `DOMAIN` puede tener ventajas si una base de datos tiene en varias tablas un campo con las mismas restricciones. Un ejemplo podría ser un campo para número de teléfono con un formato fijo en un campo de tipo `TEXT`. Si varias tablas tienen el campo número de teléfono podría ser mejor crear un `DOMAIN` en lugar de repetir las mismas restricciones en cada ocurrencia del campo. Puesto que el tipo de datos de base ya existe es posible aprovechar los mecanismos de indexación que este tiene.

`LARGE OBJECT`: Este tipo de objetos tiene dos privilegios `SELECT` y `UPDATE`, que permiten leer o escribir el objeto en cuestión.

`FOREIGN DATA WRAPPER`: Este objeto es básicamente una abstracción de un servidor externo a la base de datos. El privilegio `USAGE` le permite, al que lo tenga, crear nuevos puntos de acceso a los servidores externos que cumplan con los requisitos del objeto. Si, por ejemplo, es necesario integrar una o más tablas de un servidor con una base de datos Oracle en nuestra base de datos, el responsable del proceso de integración tendrá que tener el privilegio `USAGE`.

`TABLESPACE`: El último objeto se encuentra ya muy cerca del sistema operativo. Un `TABLESPACE` es finalmente un espacio físico en el disco duro. El único privilegio que existe es `CREATE`. Los que tienen este privilegio pueden crear bases de datos usando el `TABLESPACE` por defecto. No entramos más en detalle porque finalmente los privilegios en este contexto no tienen directamente que ver con aquellos que se necesitan para las operaciones normales de los usuarios.

6 Organizar el acceso

En el transcurso de este artículo aprendimos a utilizar los fundamentos del sistema de autorización de PostgreSQL. Posiblemente los lectores vieron interesantes oportunidades para limitar potenciales daños a una base de datos. Lo que seguramente quedó claro es que la introducción de un concepto de control de acceso puede alcanzar niveles de complejidad muy grandes. Por la multidud de detalles es posible perder la visión de conjunto. En esta sección queremos analizar algunas técnicas útiles para simplificar la gestión de los privilegios de usuarios. También volveremos a enfrentarnos con unas preguntas que quedaron abiertas en secciones anteriores.

6.1 Grupos y herencia

En la secciones que anteceden hemos mostrado cómo otorgar o revocar derechos a usuarios. En PostgreSQL usuarios y grupos son el mismo tipo de objetos y es posible otorgar un rol a otro rol, otorgando indirectamente todos los privilegios de un rol a otro. Esto es un elemento muy útil para reducir la complejidad de un sistema de privilegios. En lugar de otorgar y revocar cada privilegio a cada usuario se crean grupos y se definen los privilegios a nivel del grupo. Usuarios entonces recibirán privilegios por medio de la pertenencia a un grupo y no directamente. Esto es lo que en este contexto se llama “herencia”. Los privilegios son heredados de todos los roles al que un usuario pertenece.

En nuestro ejemplo hemos otorgado privilegios a un usuario user1. Tenemos también otro usuario user2 que posiblemente necesite los mismos privilegios. Con la solución de arriba tendríamos que otorgar todos los privilegios de user1 a user2. Es evidente que esto se puede volver complicado si hay más de dos usuarios o si los privilegios cambian con el tiempo. En lugar de modificar los privilegios de un solo grupo se tendría que modificar los de cada usario. Para implementar el uso de un grupo para los usuarios como user1 y user2 podríamos crear primero un grupo, otorgar al grupo todos los privilegios necesarios, revocar todos los privilegios de user1 y finalemente otorgar el grupo a user1 y user2. Afortunadamente PostgreSQL conoce una solución mucho más sencilla.

```
admin@uci=> ALTER ROLE user1 RENAME TO uci_users;
admin@uci=> ALTER ROLE uci_users NOLOGIN;
admin@uci=> CREATE ROLE user1 LOGIN PASSWORD 'xxx';
admin@uci=> ALTER ROLE user1 SET search_path=uci,public;
admin@uci=> GRANT uci_users TO user1;
admin@uci=> GRANT uci_users TO user2;
admin@uci=> REVOKE CONNECT ON DATABASE uci FROM user2;
admin@uci=> REVOKE USAGE ON SCHEMA uci FROM user2;
```

En lugar de crear un grupo y otorgar los privilegios modificamos el rol user1 dándole otro nombre y quitándole el atributo `LOGIN`. De esta manera hemos transformado un usario en un grupo. Por eso tenemos que volver a crear el usario y actualizar su search_path. A diferencia de los privilegios, los atributos no se heredan. Acto seguido añadimos user1 y user2 al grupo uci_users. Las últimas dos líneas revocan los privilegios que le habíamos otorgado anteriormente para nuestros tests. Un chequeo de los privilegios muestra ahora lo siguiente.

```
admin@uci=> \l uci
                                 List of databases
 Name | Owner | Encoding |   Collate   |    Ctype    | Access privileges
------+-------+----------+-------------+-------------+-------------------
 uci  | admin | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =T/admin          +
      |       |          |             |             | admin=CTc/admin   +
      |       |          |             |             | uci_users=c/admin
```

Como vemos la acción `RENAME TO` se propaga a las asociaciones de privilegios. Una característica muy agradable. De la misma manera podemos averiguar si el acceso a los demás objetos también se volvió como lo queríamos.

```
admin@uci=> \dn+ uci
                           List of schemas
  Name  |  Owner   |  Access privileges   |      Description
--------+----------+----------------------+------------------------
 uci    | admin    | admin=UC/admin      +|
        |          | uci_users=U/admin    |
```

```
admin@uci=> \dp
                                    Access privileges
 Schema |         Name          |   Type   |  Access privileges  | Column access privileges
--------+-----------------------+----------+---------------------+--------------------------
 uci    | catalogue             | table    | admin=arwdDxt/admin+| name:                   +
        |                       |          | uci_users=rd/admin  |   uci_users=aw/admin    +
        |                       |          |                     | location:               +
        |                       |          |                     |   uci_users=aw/admin    +
        |                       |          |                     | price:                  +
        |                       |          |                     |   uci_users=aw/admin
 uci    | catalogue_item_id_seq | sequence | admin=rwU/admin    +|
        |                       |          | uci_users=U/admin   |
 uci    | log_changes           | table    | admin=arwdDxt/admin+|
        |                       |          | uci_users=a/admin   |
```

Ahora vamos a reinicializar nuestras tablas y secuencias para hacer unos test con user1 y user2.

```
admin@uci=> TRUNCATE catalogue;
admin@uci=> ALTER SEQUENCE catalogue_item_id_seq RESTART;
admin@uci=> TRUNCATE log_changes ;
```

Como vemos en estos ejemplos mínimos tanto user1 como user2 tienen las mismas capacidades.

```
user1@uci=> INSERT INTO uci.catalogue (name, price) VALUES ('Nuestra América',12.90);
user2@uci=> INSERT INTO uci.catalogue (name, price) VALUES ('Il nome della rosa',21.10);
```

Nuestro catálogo tiene los datos insertados por los dos usuarios.

```
admin@uci=> SELECT * FROM uci.catalogue;
 item_id |        name        |  location  | price | responsible
---------+--------------------+------------+-------+-------------
       1 | Nuestra América    | Main store | 12.90 | user1
       2 | Il nome della rosa | Main store | 21.10 | user2
(2 rows)
```

La tabla con los registros de cambios también tiene las entradas que esperábamos.

```
admin@uci=> SELECT * FROM uci.log_changes ;
        change_date         | change_type |  change_table  | change_user
----------------------------+-------------+----------------+-------------
 2015-06-15 10:41:33.871477 | INSERT      | uci.catalogue  | user1
 2015-06-15 10:47:39.020223 | INSERT      | uci.catalogue  | user2
```

Cuando se comienza a trabajar con grupos se vuelve algo más difícil averiguar quiénes

tienen cuáles privilegios. Las funciones de sistema para averiguar privilegios incluyen los privilegios heredados de un grupo.

```
admin@uci=> SELECT has_table_privilege('user1','catalogue','select');
 has_table_privilege
---------------------
 t
```

En cambio por medio de los comandos de psql solo vemos al grupo y sus privilegios. Lo que tenemos que saber además de esto es quién pertenece a cual grupo. Mientras no se tengan muchos usuarios se puede utilizar `\du` y averiguar la columna "Member of" como aquí se muestra. El comando no permite seleccionar directamente en base al campo de las membresías.

```
admin@uci=> \du user*
            List of roles
 Role name | Attributes |  Member of
-----------+------------+-------------
 user1     |            | {uci_users}
 user2     |            | {uci_users}
```

Apenas se tengan más usuarios y grupos esta variante no es muy viable. Con una consulta del catálogo de PostgreSQL se pueden encontrar todos los miembros de un grupo.

```
SELECT rg.rolname AS group_name, rr.rolname AS role_name, gr.rolname AS grantor
FROM pg_auth_members m
JOIN pg_roles rg ON (rg.oid = m.roleid)
JOIN pg_roles rr ON (rr.oid = m.member)
JOIN pg_roles gr ON (gr.oid = m.grantor)
WHERE has_database_privilege(rr.rolname,'uci','connect')
ORDER BY rg.rolname, rr.rolname;


 group_name | role_name | grantor
------------+-----------+---------
 uci_users  | user1     | admin
 uci_users  | user2     | admin
```

Nótese que limitamos la consulta a los roles que tienen acceso a la base de datos. Con esta información sabemos al final cuáles usuarios heredan los privilegios de uci_users. La herencia de privilegios en roles es una capacidad que abre muchas posibilidades. Entre otras la posibilidad de cada usuario de otorgarse a sí mismo a otros. El efecto de esta capacidad es que cada usuario puede básicamente otorgarle el acceso a una base de datos y a todas las acciones que puede ejecutar en ella a otro usuario, sin que el responsable de la base de datos (en nuestro caso admin) se entere. Es importante que el lector sea consciente

de que esto no se limita a un nivel de indirección. Véase el ejemplo siguiente.

```
admin@uci=> CREATE ROLE user3 LOGIN PASSWORD 'xxx';
admin@uci=> CREATE ROLE user4 LOGIN PASSWORD 'xxx';


user1@uci=> GRANT user1 TO user3;
user3@uci=> GRANT user3 TO user4;
user4@uci=> INSERT INTO uci.catalogue (name,price) VALUES ('Trash entry',10000.00);
user4@uci=> SELECT * FROM uci.catalogue;
 item_id |        name        |  location  |  price   | responsible
---------+--------------------+------------+----------+-------------
       1 | Nuestra América    | Main store |    12.90 | user1
       2 | Il nome della rosa | Main store |    21.10 | user2
       3 | Trash entry        | Main store | 10000.00 | user4
```

Si el análisis de los riesgos y daños asociados lo requiere es indispensable adoptar una técnica que nos permita evitar que usuarios puedan otorgar privilegios indiscrimindadamente. Una posible medida es monitorear la tabla pg_auth_members y revocar roles que no hayan sido otorgados por roles autorizados. La consulta que vimos anteriormente ahora arroja este resultado.

```
 group_name | role_name | grantor
------------+-----------+---------
 uci_users  | user1     | admin
 uci_users  | user2     | admin
 user1      | user3     | user1
 user3      | user4     | user3
```

Suponiendo que en la base de datos solo el rol admin es autorizado a otorgar membresía en otros roles, se podrían revocar con el usuario admin todos las asociaciones en que el grantor no es admin. De forma parecida se podría p. ej. averiguar cuáles de los grupos son realmente grupos (es decir sin el atributo `LOGIN`). Esto solo funciona si la estructura de acceso está organizada de tal manera que los usuarios nunca reciben privilegios otorgados directamente (o solo aquellos que no pueden dañar). Una estrategia basada en monitoreo no es muy elegante y tampoco muy segura. Es una posibilidad sin embargo de buscar y (ojalá no) encontrar situaciones como la que mostramos arriba. Es posible que un usuario que haya otorgado su rol a otro no esté consciente de que su acción podría generar una amenaza contra la seguridad del sistema. En estos casos posiblemente se pueda solucionar con reglas organizativas e información abierta. Pero existen base de datos en las que se quiere evitar por defecto que las acciones de los usuarios se vuelvan en potenciales amenazas. Para ese fin podemos aprovechar del atributo `[INHERIT|NOHINHERIT]` del rol.

El valor por defecto de cada nuevo rol es `INHERIT` o sea de heredar todo los privilegios de los roles a los que pertenece con los efectos que acabamos de ver. Cuando un rol no

tiene puesto el atributo de herencia no recibe los privilegios del rol en los que se encuentra directamente. Para que pueda aprovechar de estos privilegios el usuario tiene que ponerse en el rol de grupo usando el comando `SET ROLE <role name>`. Dado que este comando solo se puede ejecutar en una sesión de base de datos, el rol tiene que estar conectado a una base de datos cualquiera. Esto implica que si el privilegio de conectar a una base de datos está definido en el grupo, el usuario nunca podrá conectarse, ya que nunca puede ponerse en el rol del grupo. Vamos a demostrar en la práctica lo que esto significa. Para empezar vamos a modificar todos los roles de los usuarios habituales quitándoles el atributo `INHERIT`.

```
admin@uci=> ALTER ROLE user1 NOINHERIT;
admin@uci=> ALTER ROLE user2 NOINHERIT;
admin@uci=> ALTER ROLE user3 NOINHERIT;
admin@uci=> ALTER ROLE user4 NOINHERIT;
```

Dado que el privilegio de conectarse a la base de datos está otorgado en el grupo, el intento de user1 de conectarse no funciona.

```
admin@uci=> \c - user1
Password for user user1:
FATAL:  permission denied for database "uci"
DETAIL:  User does not have CONNECT privilege.
```

Es entonces imprescindible otorgarle explícitamente el privilegio `CONNECT` a los usuarios que realmente tienen que trabajar con la base de datos. Al mismo tiempo, aunque no sea estrictamente necesario, le quitamos el mismo privilegio al grupo.

```
admin@uci=> REVOKE CONNECT ON DATABASE uci FROM uci_users;
admin@uci=> GRANT CONNECT ON DATABASE uci TO user1, user2;
```

La consulta sobre la tabla pg_auth_members muestra ahora que solo user1 y user2 pueden conectarse a uci. Nótese que las demás configuraciones hechas por user1 a user3 y user3 a user4 siguen presente, pero no aparecen porque user3 y user4 no tienen el privilegio de conectarse a uci ya sea directa o indirectamente.

```
 group_name | role_name | grantor
------------+-----------+---------
 uci_users  | user1     | admin
 uci_users  | user2     | admin
```

Nuestro usuario user1 puede conectarse a uci porque tiene otorgado directamente el privilegio correspondiente. El problema ahora es que los privilegios de uci_users ya no se heredan automáticamente y, por ende, user1 no puede hacer nada antes de ponerse el rol del grupo.

```
user1@uci=> SELECT * FROM catalogue;
ERROR:  relation "catalogue" does not exist
user1@uci=> SET ROLE uci_users;

user1@uci=> SELECT * FROM catalogue;
 item_id |        name        |  location  |  price   | responsible
---------+--------------------+------------+----------+-------------
       1 | Nuestra América    | Main store |    12.90 | user1
       2 | Il nome della rosa | Main store |    21.10 | user2
       3 | Trash entry        | Main store | 10000.00 | user4
```

Según el resultado de la consulta a pg_auth_members user4 (p. ej.) no puede conectarse a uci. Sabemos también que user4 pueder tomar el rol user1.

```
user1@uci=> \c - user4
Password for user user4:
FATAL:  permission denied for database "uci"
DETAIL:  User does not have CONNECT privilege.
```

Como esperábamos user4 no se puede conectar a uci. Pongamos por hipótesis que user4 pueda conectarse a otra base de datos en el clúster.

```
charles@uci=# GRANT CONNECT ON DATABASE charles TO user4;
charles@uci=> \c charles user4
user4@charles=> SET ROLE user1;
user4@charles=> \c uci user4
FATAL:  permission denied for database "uci"
DETAIL:  User does not have CONNECT privilege.
```

También en este caso user4 no puede conectarse a uci aunque pueda tomar el rol de user1. El alcance de `SET ROLE` es la sesión. Al conectarse a otra base de datos se establece una nueva sesión por lo que el `SET ROLE` es invalidado. La única manera por la que user4 pudiera conectarse a uci es por medio del privilegio correspondiente otorgado por admin o un superuser.

```
admin@uci=> GRANT CONNECT ON DATABASE uci TO user4;
user4@uci=> SET ROLE uci_users;
user4@uci=> SELECT * FROM uci.catalogue;
 item_id |        name        |  location  |  price   | responsible
---------+--------------------+------------+----------+-------------
       1 | Nuestra América    | Main store |    12.90 | user1
       2 | Il nome della rosa | Main store |    21.10 | user2
       3 | Trash entry        | Main store | 10000.00 | user4
```

El uso de esta técnica para controlar el acceso permite también mejoras en nuestro registro de actividades en la tabla log_changes. En una sesión de base de datos en PostgreSQL

se diferencia entre SESSION_USER y CURRENT_USER. En los casos normales estos dos usuarios son uno y lo mismo. Después de haber ejecutado SET ROLE el rol en que el usuario se ha puesto es reflejado en CURRENT_USER. De esta manera es posible registrar no solo quien ha hecho que actividad sino que también en que rol se encontraba el usuario cuando ejecutó la acción.

```
admin@uci=> TRUNCATE uci.log_changes;
admin@uci=> ALTER TABLE uci.log_changes ADD COLUMN change_group TEXT
admin@uci-> NOT NULL DEFAULT CURRENT_USER;
```

No es necesario hacer cambios al código de la función disparadoras porque la columna que añadimos utiliza el valor por defecto de CURRENT_USER.

```
admin@uci=> INSERT INTO uci.catalogue (name, price) VALUES ('Admin is doing stuff',0.00);
user1@uci=> SET ROLE uci_users;
user1@uci=> INSERT INTO uci.catalogue (name, price) VALUES ('user1 is doing stuff',0.00);
admin@uci=> SELECT * FROM uci.log_changes ;
        change_date         | change_type |  change_table  | change_user | change_group
----------------------------+-------------+----------------+-------------+--------------
 2015-06-15 13:52:23.809557 | INSERT      | uci.catalogue  | admin       | admin
 2015-06-15 13:52:57.858291 | INSERT      | uci.catalogue  | user1       | uci_users
```

6.2 Vistas y funciones security definer

En una base de datos, las vistas son un medio muy eficaz para esconder la complejidad de las estructuras subyacentes. Desde un punto de vista del usuario una vista y una tabla parecen lo mismo. Ambas retornan una tabla que puede ser usada en la funcionalidad de la aplicación. En PostgreSQL, en alguna circunstancias, vistas pueden utilizarse para ejecutar operaciones INSERT, UPDATE y DELETE. Aunque esta capacidad puede ser muy útil hay que tener cuidado, sobre todo cuando se usan las variantes de GRANT que permiten otorgar todos los privilegios sobre todas las tablas de un esquema. En este contexto las vistas son interpretadas como tablas. Es posible pues que se otorguen privilegios para modificar vistas, sin que eso sea realmente lo que se quiere alcanzar.

A parte de eso vistas pueden ser muy útiles para controlar el acceso del usuario sobre tablas a nivel de filas. En el ejemplo que utilizamos en este artículo le otorgamos a los usuarios user1 y user2 el privilegio SELECT sobre la tabla catalogue por medio del grupo uci_users. Si quisiéramos limitar el acceso de estos usuarios solo a las filas de las que son responsables tendríamos que poner la restricción en la selección en la aplicación. Pero esto también se puede hacer en la base de datos usando vistas. Una vista que no hace más que retornar la misma tabla seleccionando algunas filas es actualizable, es decir que se pueden

utilizar las instrucciones SQL para modificar datos sobre la vista como si fuera una tabla.

```
CREATE VIEW uci.v_catalogue AS
SELECT * FROM uci.catalogue
WHERE responsible = SESSION_USER;


admin@uci=> REVOKE SELECT ON uci.catalogue FROM uci_users;
admin@uci=> GRANT SELECT ON uci.v_catalogue TO uci_users;


user1@uci=> SET ROLE uci_users;
user1@uci=> SELECT * FROM uci.catalogue ;
ERROR:  permission denied for relation catalogue
user1@uci=> SELECT * FROM uci.v_catalogue ;
 item_id |         name         |  location  | price | responsible
---------+----------------------+------------+-------+-------------
       1 | Nuestra América      | Main store | 12.90 | user1
       5 | user1 is doing stuff | Main store |  0.00 | user1

user2@uci=> SET ROLE uci_users;
user2@uci=> SELECT * FROM uci.v_catalogue ;
 item_id |        name        |  location  | price | responsible
---------+--------------------+------------+-------+-------------
       2 | Il nome della rosa | Main store | 21.10 | user2
```

De esta manera hemos logrado que un usuario solo vea las filas de las que es responsable. Quitando el privilegio `SELECT` sobre la tabla catalogue de uci_users, le quitamos indirectamente también los privilegios para modificar filas a todos los miembros de uci_users.

```
user1@uci=> SELECT * from v_catalogue ;
 item_id |         name         |  location  | price | responsible
---------+----------------------+------------+-------+-------------
       1 | Nuestra América      | Main store | 12.90 | user1
       5 | user1 is doing stuff | Main store |  0.00 | user1
user1@uci=> UPDATE uci.catalogue SET price = 18.50 WHERE item_id = 5;
ERROR:  permission denied for relation catalogue
```

De hecho todos los usuarios del grupo siguen teniendo el privilegio para actualizar y borrar filas, pero estos privilegios requieren implícitamente el privilegio de seleccionar (es decir buscar) las filas sobre las que se tiene que ejecutar la operación. Lo que tenemos que hacer es pasar los privilegios de la tabla a la vista.

```
REVOKE INSERT (name,location,price),
       UPDATE (name,location,price),
       DELETE ON uci.catalogue FROM uci_users;
```

```
GRANT INSERT (name,location,price),
      UPDATE (name,location,price),
      DELETE ON uci.v_catalogue TO uci_users;
```

De esta manera cada usuario solo puede modificar la tabla catalogue por medio de la vista v_catalogue lo que al fin y al cabo quiere decir que solo puede operar sobre filas por las que es responsable.

```
user1@uci=> SELECT * FROM uci.v_catalogue;
 item_id |         name         |  location  | price | responsible
---------+----------------------+------------+-------+-------------
       1 | Nuestra América      | Main store | 12.90 | user1
       5 | user1 is doing stuff | Main store | 18.50 | user1

user1@uci=> INSERT INTO uci.v_catalogue (name, price) VALUES ('Insert from view',10.00);
INSERT 0 1


user1@uci=> SELECT * FROM uci.v_catalogue;
 item_id |         name         |  location  | price | responsible
---------+----------------------+------------+-------+-------------
 [...]
       8 | Insert from view     | Main store | 10.00 | user1

user1@uci=> UPDATE uci.v_catalogue SET price = 5.00 WHERE item_id = 8;
UPDATE 1
user1@uci=> DELETE FROM uci.v_catalogue WHERE item_id = 8;
DELETE 1
```

El usuario user1 no va a poder hacer cambios sobre filas de otro usuario, porque la búsqueda de la fila a cambiar no regresará ningún elemento.

```
admin@uci=> SELECT * FROM uci.catalogue WHERE responsible <> 'user1';
 item_id |         name         |  location  |  price   | responsible
---------+----------------------+------------+----------+-------------
 [...]
       7 | Trash entry          | Main store | 10000.00 | user4

user1@uci=> DELETE FROM uci.v_catalogue WHERE item_id = 7;
DELETE 0
```

Cabe mencionar que habiéndole quitado los privilegios para modificar datos de la tabla directamente, un intento en esa dirección llevaría a un error en tiempo de ejecución (permission denied). La situación actual de los privilegios se muestra aquí enseguida.

```
admin@uci=> \dp
 Schema |         Name          |   Type   |  Access privileges  | Column access privileges
--------+-----------------------+----------+---------------------+-------------------------
 uci    | catalogue             | table    | admin=arwdDxt/admin |
 uci    | log_changes           | table    | admin=arwdDxt/admin+|
        |                       |          | uci_users=a/admin   |
 uci    | v_catalogue           | view     | admin=arwdDxt/admin+| name:                    +
        |                       |          | uci_users=rd/admin  |   uci_users=aw/admin     +
        |                       |          |                     | location:                +
        |                       |          |                     |   uci_users=aw/admin     +
        |                       |          |                     | price:                   +
        |                       |          |                     |   uci_users=aw/admin
```

Nos queda todavía el problema de la tabla log_changes que almacena el registro de cambios hechos por los usuarios sobre la tabla catalogue. Para poder utilizar la función disparadora tuvimos que otorgarle al grupo uci_users el privilegio `INSERT` sobre log_changes. El problema generado es que de esta forma cada miembro del grupo puede insertar filas en la tabla log_changes sin pasar por el trigger. Para solucionar este problema podemos recurrir a funciones security definer. Una función definida de esta forma es ejecutada con los privilegios del proprietario de la función. Al utilizar esta opción hay que tener mucho cuidado. Por defecto el privilegio `EXECUTE` es otorgado a todos los usuarios. En el caso de una función disparadora no se ponen problemas porque no es posible llamar la función directamente. Pero cualquier función podría ser declarada con la opción security definer para permitir a ciertos usuarios normales ejecutar operaciones por las que no tendrían privilegios normalmente, como es el caso también de nuestra función disparadora. El objetivo es que el grupo uci_users ya no tenga la posibilidad de insertar directamente filas en la tabla log_changes. Para eso volvemos a crear la función disparadora como sigue.

```
CREATE OR REPLACE FUNCTION uci.log_changes()
RETURNS TRIGGER
AS $$
BEGIN
  -- Same body as before
END;
$$ LANGUAGE plpgsql
SECURITY DEFINER;
```

Nótese que la única diferencia es la opción `SECURITY DEFINER` al final de la declaración. Una vez que la función sea disponible, aunque para este tipo de funciones no sea necesario, vamos a limitar el privilegio de ejecutarla al grupo uci_users.

```
admin@uci=> REVOKE EXECUTE ON FUNCTION uci.log_changes() FROM PUBLIC;
admin@uci=> GRANT EXECUTE ON FUNCTION uci.log_changes() TO uci_users;
admin@uci=> REVOKE INSERT ON uci.log_changes FROM uci_users;
```

Ahora user1 como miembro de uci_users ya no va a poder insertar datos en log_changes directamente. El trigger, en cambio funciona normalmente y como es esperado.

```
user1@uci=> INSERT INTO uci.log_changes (change_date, change_type,
user1@uci->                                change_table, change_user, change_group)
user1@uci-> VALUES (CURRENT_DATE - 180, 'DELETE', 'uci.catalogue', 'user2', 'user2');
ERROR:  permission denied for relation log_changes
user1@uci=> INSERT INTO uci.v_catalogue (name, price) VALUES ('Tractatus', 15.60);


admin@uci=> SELECT * FROM uci.catalogue;
 item_id |           name            |  location  |  price   | responsible
---------+---------------------------+------------+----------+-------------
 [...]
      11 | Tractatus                 | Main store |    15.60 | user1
(7 rows)

admin@uci=> SELECT * FROM uci.log_changes;
        change_date         | change_type |  change_table  | change_user | change_group
----------------------------+-------------+----------------+-------------+--------------
 [...]
 2015-06-15 16:21:41.922081 | INSERT      | uci.catalogue  | user1       | admin
```

Como registrado en log_changes el grupo (o sea `CURRENT_USER`) es admin. La razón es que la función es ejecutada como si el usario fuera admin. De esta forma hemos logrado que la tabla log_changes solo pueda ser actualizada por medio del trigger sobre la tabla catalogue, a parte de su propietario o por un superuser, quienes pueden manipular los datos de la tabla log_changes por defecto.

6.3 Privilegios por defecto

En varias ocasiones, anteriormente, vimos que algunos privilegios son otorgados por defecto en el momento en que se crea un objeto. Lo más obvio y no requiere más comentario es que el proprietario de un objeto tiene todos los privilegios sobre el mismo. Un superuser no está sujeto al sistema de autorización y por lo tanto tiene todos los privilegios sobre todos los objetos en un clúster. Cuando se crean algunos objetos, PostgreSQL otorga por defecto algunos privilegios a `PUBLIC`. En la documentación oficial de PostgreSQL estos se encuentran en los comentarios sobre `GRANT` ([Pos15a], p. 1523):

- `CONNECT` y `TEMPORARY` sobre bases de datos.
- `EXECUTE` sobre funciones.
- `USAGE` sobre lenguajes.

Hemos visto las posibilidades para modificar estos privilegios por defecto, cuando los objetos han sido creados. Este es también el entorno habitual de una base de datos productiva. Existen sin embargo situaciones en las que se quiere poder otorgar ciertos derechos automáticamente. Para eso PostgreSQL conoce el comando `ALTER DEFAULT PRIVILEGES`.

Si, por ejemplo, una base de datos existe en una instancia test y está en desarrollo, puede ser que se quiera dejarle la oportunidad a los desarrolladores de otorgarse privilegios entre sí automáticamente. Por supuesto el usuario que crea un objeto puede otorgar privilegios sobre él a otros. Lo más probable sigue siendo que este tipo de detalles se olviden. Para nuestro ejemplo vamos a suponer que el equipo de desarrollo de la interfaz web pública quiera unas vistas para visualizar informaciones en el sitio. Ya que todo la interfaz pública está todavía en discusión el equipo pide constantemente nuevas vistas o modifica las que ya existen. Puesto que no es posible modificar la estructura de fila retornada por una vista, el efecto es que estos cambios implican destruir la vista y crear una nueva. Nuestro usuario admin decidió que las vistas para acceso público pueden ser ubicadas en el esquema public, sobre el que `PUBLIC` tiene el privilegio `USAGE`.

```
admin@uci=> \dn+ public
                          List of schemas
  Name  |  Owner   |  Access privileges   |      Description
--------+----------+----------------------+------------------------
 public | postgres | postgres=UC/postgres+| standard public schema
        |          | =U/postgres         +|
        |          | admin=UC/postgres    |
```

En casos normales admin tendría que crear la vista y otorgar posteriormente el privilegio `SELECT` al usuario para la aplicación web. Alternativamente y si las informaciones retornadas por la vista son realmente públicas admin puede otorgarle el privilegio a `PUBLIC`.

```
admin@uci=> CREATE VIEW public.v_catalogue_public AS
admin@uci-> SELECT name, price FROM uci.catalogue;
admin@uci=> GRANT SELECT ON public.v_catalogue_public TO PUBLIC;
```

```
admin@uci=> \dp v_catalogue_public
                                  Access privileges
 Schema |        Name        | Type |  Access privileges  | Column access privileges
--------+--------------------+------+---------------------+--------------------------
 public | v_catalogue_public | view | admin=arwdDxt/admin+|
        |                    |      | =r/admin            |
```

En lugar de tener cada vez que otorgar el privilegio admin podría definir un privilegio por defecto. En este caso con el código siguiente.

```
admin@uci=> ALTER DEFAULT PRIVILEGES FOR ROLE admin IN SCHEMA public
admin@uci-> GRANT SELECT ON TABLES TO PUBLIC;
```

Los privilegios por defecto que existen en la base de datos se pueden ver con el comando `\ddp`.

```
admin@uci=> \ddp
         Default access privileges
 Owner | Schema | Type  | Access privileges
-------+--------+-------+-------------------
 admin | public | table | =r/admin
(1 row)
```

En el output se ve para cuál esquema se tiene que aplicar el privilegio (vacío si el privilegio por defecto se tiene que aplicar para toda la base de datos), sobre qué tipo de objeto (nótese que tablas y vistas son considerada iguales) y finalmente cuál privilegio y para quién. Una vez definido el privilegio por defecto, si admin crea una vista nueva el privilegio será otorgado automáticamente.

```
admin@uci=> CREATE VIEW public.v_catalogue_locations AS
admin@uci-> SELECT location FROM uci.catalogue;


admin@uci=> \dp v_catalogue_*
                                   Access privileges
 Schema |         Name          | Type |  Access privileges  | Column access privileges
--------+-----------------------+------+---------------------+--------------------------
 public | v_catalogue_locations | view | =r/admin           +|
        |                       |      | admin=arwdDxt/admin |
 public | v_catalogue_public    | view | admin=arwdDxt/admin+|
        |                       |      | =r/admin            |
```

Para borrar un privilegio por defecto no existe un comando `DROP` como es constumbre para objetos. Lo que hay que hacer en cambio es modificar el privilegio por defecto requiriendo exactamente lo opuesto. El privilegio por defecto que admin creó anterioremente

fue “Si admin crea una tabla o una vista en el esquema public otorga SELECT a PUBLIC”. Entonces el revés será “Si admin crea una tabla o una vista en el esquema public revoca SELECT de PUBLIC”.

```
admin@uci=> ALTER DEFAULT PRIVILEGES FOR ROLE admin IN SCHEMA public
admin@uci-> REVOKE SELECT ON TABLES FROM PUBLIC;
admin@uci=> \ddp
         Default access privileges
 Owner | Schema | Type | Access privileges
-------+--------+------+-------------------
(0 rows)


admin@uci=> CREATE VIEW public.v_catalogue_responsibles AS
admin@uci-> SELECT responsible FROM uci.catalogue;
admin@uci=> \dp v_catalogue_responsibles
                                    Access privileges
 Schema |           Name           | Type | Access privileges | Column access privileges
--------+--------------------------+------+-------------------+--------------------------
 public | v_catalogue_responsibles | view |                   |
```

Cabe mencionar que si se altera un privilegio por defecto de tal manera que el resultado corresponda con uno que ya existe no se generan entradas.

```
admin@uci=> ALTER DEFAULT PRIVILEGES FOR ROLE admin
admin@uci-> GRANT EXECUTE ON FUNCTIONS TO PUBLIC;


admin@uci=> \ddp
         Default access privileges
 Owner | Schema | Type | Access privileges
-------+--------+------+-------------------
(0 rows)
```

El comando no tiene efecto porque en PostgreSQL ya está definido por defecto que al crear una función se le otorgue el privilegio EXECUTE a PUBLIC.

Nótese que al otorgar un privilegio sobre una vista, indirectamente se otorga el mismo privilegio sobre la tabla subyacente. En la vista calificamos la tabla con el nombre del esquema para evitar que usuarios reciban el error “does not exists” por falta de esquema en su search_path. PUBLIC no tiene ningún privilegio sobre el esquema uci o sobre sus objetos. Sin embargo puede seleccionar filas.

```
admin@uci=> CREATE ROLE user_public LOGIN PASSWORD 'xxx';
admin@uci=> GRANT CONNECT ON DATABASE uci TO user_public;
```

```
user_public@uci=> SELECT * FROM v_catalogue_public;
        name        | price
--------------------+-------
 Nuestra América    | 12.90
 Il nome della rosa | 21.10
 Tractatus          | 15.60
```

Esta característica puede tener efectos inesperados. Si, por ejemplo, le otorgamos a PUBLIC el privilegio INSERT sobre una vista, le otorgamos indirectamente como vimos el mismo privilegio sobre la tabla subyacente. En nuestro ejemplo si otorgáramos INSERT sobre la vista v_catalogue_public a PUBLIC, todo el mundo podría teóricamente insertar filas en la tabla uci.catalogue. Concretamente eso no sucede, porque la tabla tiene una secuencia sobre la que el privilegio USAGE se debería de otorgar explícitamente. De no ser el caso, sin embargo, el efecto de INSERT sobre una vista sería precisamente lo mismo que otorgar el privilegio directamente sobre la tabla[1].

```
admin@uci=> CREATE TABLE uci.catalogue_copy AS SELECT * FROM uci.catalogue;
admin@uci=> CREATE VIEW public.v_catalogue_copy_public AS
admin@uci-> SELECT name, price FROM uci.catalogue_copy;


admin@uci=> GRANT SELECT, INSERT ON public.v_catalogue_copy_public TO PUBLIC;

user_public@uci=> INSERT INTO v_catalogue_copy_public (name, price)
user_public@uci-> VALUES ('From public',90.00);
INSERT 0 1

admin@uci=> SELECT * FROM uci.catalogue_copy;
 item_id |        name        |  location  | price | responsible
---------+--------------------+------------+-------+-------------
 [...]
         | From public        |            | 90.00 |
```

Los privilegios por defecto son útiles si se quiere forzar una configuración diferente de la que PostgreSQL ofrece por defecto. Como último ejemplo vamos a mostrar como se podría utilizar este mecanismo para controlar el uso de nuevas funciones. Queremos limitar el uso de funciones en toda la base de datos al grupo uci_users.

[1]Como relatado en el texto si existen restricciones sobre la tabla, como el uso de secuencias o campos que no pueden estar vacíos un intento de insertar datos acabaría con un error. La copia de la tabla no tiene ninguna de las restricciones de la tabla original.

```
admin@uci=> ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
admin@uci=> ALTER DEFAULT PRIVILEGES FOR ROLE admin GRANT EXECUTE ON FUNCTIONS TO uci_users;
admin@uci=> \ddp
           Default access privileges
 Owner | Schema |   Type   | Access privileges
-------+--------+----------+-------------------
 admin |        | function | admin=X/admin    +
       |        |          | uci_users=X/admin
```

Nótese que la expresión `admin=X/admin` fue creada con el primer comando para revocar el privilegio de `PUBLIC`. Para el test de los privilegios por defecto volvemos a crear las funciones show_schema que utilizamos en la sección sobre el acceso a funciones usando esta vez distintos nombres para identificarlas mejor.

```
DROP FUNCTION uci.show_schema();
DROP FUNCTION public.show_schema();

CREATE FUNCTION uci.show_schema_uci()
[...]
CREATE FUNCTION public.show_schema_public()
[...]
```

Después de haber recreado las funciones averiguamos los privilegios puestos automáticamente.

```
admin@uci=> SELECT proname, proacl FROM pg_proc WHERE proname LIKE 'show_schema_%';
      proname       |              proacl
--------------------+-----------------------------------
 show_schema_public | {admin=X/admin,uci_users=X/admin}
 show_schema_uci    | {admin=X/admin,uci_users=X/admin}
```

Precisamente como se ha indicado en los privilegios por defecto, ahora solo los miembros de uci_users, el proprietario y superusers pueden ejecutar ambas funciones.

```
user1@uci=> SET ROLE uci_users;
user1@uci=> SELECT * FROM uci.show_schema_uci();
-[ RECORD 1 ]----+--------------------
show_schema_uci  | I am in schema uci


user1@uci=> SELECT * FROM public.show_schema_public();
-[ RECORD 1 ]------+----------------------
show_schema_public | I am in schema public
```

Nuestro user_public para la interfaz web no puede ejecutar ninguna, tampoco la que se encuentra en el esquema public.

```
user_public@uci=> SELECT * FROM uci.show_schema_uci();
ERROR:  permission denied for schema uci
user_public@uci=> SELECT * FROM public.show_schema_public();
ERROR:  permission denied for function show_schema_public
```

6.4 Seguridad a nivel de filas

Desde la versión 9.5, PostgreSQL introdujo un mecanismo de control de acceso a nivel de filas, llamado "Row Level Security". La sección 5.7 de la documentación oficial en red se dedica a esta nueva capacidad: [Pos15b]. El objetivo de la seguridad a nivel de filas es poder definir condiciones que controlan la visibilidad de las filas de una tabla, pudiéndose imaginar este mecanismo como un filtro. En un capítulo anterior logramos este fin usando una vista para seleccionar filas por medio de la palabra clave `WHERE`. La ventaja de la seguridad a nivel de filas en PostgreSQL es que no es necesario crear una o más vistas para cada tabla, sino que las reglas se definen directamente sobre la tabla.

En el siguiente ejemplo repetimos lo que hicimos anteriormente por medio de una vista, utilizando la seguridad a nivel de filas en su lugar. Para empezar borramos la vista y volvemos a otorgar los privilegios directamente sobre la tabla.

```
DROP VIEW operations.v_catalogue;
GRANT SELECT, DELETE ON catalogue TO uci_users;
GRANT INSERT (name, location, price),
      UPDATE (name, location, price) ON catalogue TO uci_users;
```

Ahora podemos crear una policy (básicamente una norma) sobre la tabla para el grupo uci_users. Puesto que la condición para seleccionar y modificar filas es la misma, podemos crear una norma para todos los comandos.

```
CREATE POLICY uci_users_policy ON operations.catalogue
FOR ALL
TO uci_users
USING (responsible = SESSION_USER)
WITH CHECK (responsible = SESSION_USER);
```

Con la palabra clave `USING` definimos las reglas que se tienen que seguir cuando se busquen datos, incluidos aquellos para modificar o borrar filas. Como se ve en el ejemplo, la condición es exactamente igual a la que utilizamos en la vista. Para controlar operaciones que crean nuevas filas se utiliza la palabra clave `WITH CHECK`. La creación de una nueva fila no es lo mismo que añadir una nueva fila en la tabla. En el caso de un `UPDATE` PostgreSQL

genera una nueva fila y marca la "vieja" como sucia para que sea borrada al ejecutar un `vacuum`.

Para averiguar este hecho podemos utilizar la columna de sistema que indica donde se encuentra la fila físicamente en la tabla.

```
user1@uci=> SELECT ctid, * from operations.catalogue WHERE item_id = 9;
 ctid  | item_id |   name   |  location  | price | responsible
-------+---------+----------+------------+-------+-------------
 (0,7) |       9 | test 2   | Main store |  0.00 | user1

user1@uci=> UPDATE operations.catalogue SET name = 'Book three' WHERE item_id = 9;
UPDATE 1
user1@uci=> SELECT ctid, * from operations.catalogue WHERE item_id = 9;
  ctid  | item_id |    name    |  location  | price | responsible
--------+---------+------------+------------+-------+-------------
 (0,12) |       9 | Book three | Main store |  0.00 | user1
```

Una vez que la policy está disponible se tiene que activar para que tenga efecto. Eso se efectúa por medio del comando `ALTER TABLE`.

```
ALTER TABLE operations.catalogue ENABLE ROW LEVEL SECURITY;
```

Como siempre cuando se implementan normas de seguridad, es indispensble averigurar que todo funcione como se espera. Por eso nos conectamos a la base de datos con el usuario user1 y ejecutamos unos cuanta operaciones.

```
user1@uci=> SELECT * FROM operations.catalogue;
 item_id |   name   |  location  | price | responsible
---------+----------+------------+-------+-------------
       1 | Book one | Main store | 10.00 | user1
       2 | Book two | Main store | 12.40 | user1

user1@uci=> INSERT INTO operations.catalogue (name, price)
user1@uci-> VALUES ('Historia de Cuba',15.10);
user1@uci=> SELECT * FROM operations.catalogue;
 item_id |       name       |  location  | price | responsible
---------+------------------+------------+-------+-------------
       1 | Book one         | Main store | 10.00 | user1
       2 | Book two         | Main store | 12.40 | user1
       8 | Historia de Cuba | Main store | 15.10 | user1


user1@uci=> UPDATE operations.catalogue SET price = 9.10
user1@uci-> WHERE item_id = 8;
```

```
user1@uci=> SELECT * FROM operations.catalogue;
 item_id |       name       |  location  | price | responsible
---------+------------------+------------+-------+-------------
       1 | Book one         | Main store | 10.00 | user1
       2 | Book two         | Main store | 12.40 | user1
       8 | Historia de Cuba | Main store |  9.10 | user1

user1@uci=> INSERT INTO operations.catalogue (name, price, responsible)
user1@uci-> VALUES ('Historia de Francia',18.60,'user2');
ERROR:  permission denied for relation catalogue
```

Hasta este punto, el comportamiento del sistema corresponde a lo que queríamos alcanzar. Nótese que el último error es originado de los privilegios sobre las columnas y no de la policy. En los casos de `UPDATE` y `DELETE`, ya que `SELECT` devolverá solo las filas del usuario que está conectado, no recibiremos errores y no habrán cambios en los datos. Es decir tal y cual como lo habíamos realizado utilizando la vista. Si el usuario intenta, p. ej., borrar una fila que no le pertenece, no lo va a lograr por la simple razón, que la fila no es visible para él.

```
admin@uci=> SELECT * FROM operations.catalogue;
                              -
 item_id |       name       |  location  | price | responsible
---------+------------------+------------+-------+-------------
       1 | Book one         | Main store | 10.00 | user1
       2 | Book two         | Main store | 12.40 | user1
       7 | Book three       | Main store | 11.20 | user2 <- intentamos borrar esta fila
       8 | Historia de Cuba | Main store |  9.10 | user1 <- intentamos borrar esta fila

user1@uci=> DELETE FROM operations.catalogue WHERE item_id = 7;
DELETE 0
user1@uci=> DELETE FROM operations.catalogue WHERE item_id = 8;
DELETE 1
```

Las normas que existen sobre una tabla se pueden ver usando el comando `\dp`. En el texto mostramos solo la parte que se refiere a las normas por razones de espacio en la cuartilla.

```
admin@uci=> \dp
                                 Access privileges
    Name   |                     Policies
-----------+-------------------------------------------------
 catalogue | uci_users_policy:                               +
           |   (u): (responsible = ("session_user"())::text)+
           |   (c): (responsible = ("session_user"())::text)+
           |   to: uci_users
```

También se ven las normas en la descripción de la tabla (\d).

```
admin@uci=> \d operations.catalogue
[...]
Policies:
    POLICY "uci_users_policy" FOR ALL
      TO uci_users
      USING (responsible = ("session_user"())::text)
      WITH CHECK (responsible = ("session_user"())::text)
[...]
```

En la descripción de la tabla se ve también si la seguridad a nivel de filas es activada o no. Si no fuera activada, veríamos en el encabezado de la sección correspondiente la siguiente información.

```
[...]
Policies (Row Security Disabled):
    POLICY "uci_users_policy" FOR ALL [...]
```

Finalmente es posible también consultar la nueva vista pg_policies.

```
admin@uci=> select * from pg_policies;
-[ RECORD 1 ]----------------------------------------
schemaname | operations
tablename  | catalogue
policyname | uci_users_policy
roles      | {uci_users}
cmd        | ALL
qual       | (responsible = ("session_user"())::text)
with_check | (responsible = ("session_user"())::text)
```

En nuestro ejemplo tenemos un caso muy sencillo en el que la condición para todos los comandos es la misma. En la realidad pueden haber casos mucho más complejos con condiciones diferentes para distintos comandos. Supongamos que nuestros usuarios deban tener la posibilidad de ver todas las filas pero, como antes, sólo de modificar filas de las que son los responsables. En este caso ya no es posible utilizar una sola norma para todos los comandos porque la palabra clave USING define la condición a utilizarse para SELECT, UPDATE y DELETE. Lo que queremos alcanzar es que el filtro para el SELECT permita al usuario ver todas las filas, mientras que el filtro para los otros comandos siga siendo el nombre del usuario. En el caso del INSERT la condición se expresa en WITH CHECK.

Entonces tenemos que borrar la norma actual y crear una por cada comando.

```
DROP POLICY uci_users_policy ON operations.catalogue;
```

```
CREATE POLICY uci_select ON operations.catalogue
FOR SELECT TO uci_users
USING (true);
```

Este caso es sencillo. La prueba para implementar la norma cuando un miembro de uci_users ejecuta un SELECT siempre retorna true. De esta forma la norma dice "devuelve todas las filas para usuarios que tengan el privilegio SELECT sobre la tabla operations.catalogue y sean miembros del grupo uci_users". Para este comando la opción WITH CHECK no tiene sentido y no es permitida. Un intento de añadir esta condición retorna un error.

```
ERROR: WITH CHECK cannot be applied TO SELECT or DELETE.
```

Para usuarios que no son miembros del grupo uci_users no existe una norma definida y, por lo tanto, siempre recibirán un conjunto vacío.

```
CREATE POLICY uci_insert ON operations.catalogue
FOR INSERT TO uci_users
WITH CHECK (responsible = SESSION_USER);
```

En el caso de INSERT sólo se puede usar la opción WITH CHECK. En este caso usuarios que no pertenecen al grupo uci_users y tienen el privilegio de añadir filas, recibirán un mensaje de violación de la norma vigente si intentan ejecutar esta operación.

```
ERROR: new row violates row level security policy for ‘‘catalogue’’.
```

UPDATE permite tanto USING como WITH CHECK, aunque si se omite esta última la condición definida en USING es implícitamente usada para ambas opciones.

```
CREATE POLICY uci_update ON operations.catalogue
FOR UPDATE TO uci_users
USING (responsible = SESSION_USER);
```

Por medio de la opción USING los miembros del grupo uci_users solo pueden modificar las filas en las que su nombre de usuario aparece en la columna responsible. Esto funciona porque el comando UPDATE solo ve las filas que cumplen con la condición. Dado que no mencionamos en la creación de la norma ningúna opción WITH CHECK, PostgreSQL usará por defecto la misma condición definida en USING, al insertar filas. La diferencia con respecto al caso de INSERT es que la norma toma efecto cuando la fila se crea por medio de un UPDATE como se mostró al comienzo de este capítulo.

En el último caso, para el DELETE solo se puede definir la opción USING.

```
CREATE POLICY uci_delete ON operations.catalogue
FOR DELETE TO uci_users
USING (responsible = SESSION_USER);
```

Una vez que la configuración de todos los casos necesarios está implementada, tenemos que hacer unas cuantas pruebas para averiguar que todo funcione como queremos. Para eso tenemos que volver a poner los privilegios como estaban antes del experimento con `UPDATE`, es decir con los privilegios `SELECT` y `DELETE` sobre la tabla e `INSERT` y `UPDATE` sobre las columnas name, location y price. Las normas definidas se ven en la siguiente tabla.

```
admin@uci=> \dp operations.catalogue


   Name    |                    Policies
-----------+-------------------------------------------------
 catalogue | uci_select (r):                                +
           |   (u): true                                    +
           |   to: uci_users                                +
           | uci_insert (a):                                +
           |   (c): (responsible = ("session_user"())::text)+
           |   to: uci_users                                +
           | uci_update (w):                                +
           |   (u): (responsible = ("session_user"())::text)+
           |   to: uci_users                                +
           | uci_delete (d):                                +
           |   (u): (responsible = ("session_user"())::text)+
           |   to: uci_users
```

Los miembros de uci_users tienen que poder leer todas las filas, pero solo modificar aquellas por las que son responsables.

```
user1@uci=> SELECT * FROM operations.catalogue;
 item_id |    name    |  location  | price | responsible
---------+------------+------------+-------+-------------
       1 | Book one   | Main store | 10.00 | user1
       2 | Book two   | Main store | 12.40 | user1
       7 | Book three | Main store | 11.20 | user2

user1@uci=> UPDATE operations.catalogue SET price = 8.50 WHERE item_id = 7;
UPDATE 0
user1@uci=> UPDATE operations.catalogue SET price = 8.50 WHERE item_id = 1;
UPDATE 1

user1@uci=> DELETE FROM operations.catalogue WHERE item_id = 7;
DELETE 0
```

```
user1@uci=> DELETE FROM operations.catalogue WHERE item_id = 2;
DELETE 1
```

Además pueden añadir filas, siempre y cuando no especifiquen un responsable diferente de ellos mismos.

```
user1@uci=> INSERT INTO operations.catalogue (name, price)
user1@uci-> VALUES ('Book four',8.50);
INSERT 0 1


user1@uci=> INSERT INTO operations.catalogue (name, price, responsible)
user1@uci-> VALUES ('Book five',4.50,'user2');
ERROR:  permission denied for relation catalogue
```

El resultado de las operaciones efectuadas confirma que los privilegios y las normas están puestos de manera correspondiente al concepto de seguridad que queremos implementar.

```
user1@uci=> SELECT * FROM operations.catalogue;
 item_id |    name    |  location  | price | responsible
---------+------------+------------+-------+-------------
       1 | Book one   | Main store |  8.50 | user1
       7 | Book three | Main store | 11.20 | user2
       8 | Book four  | Main store |  8.50 | user1
```

Podemos averiguar también si la norma protege la tabla contra usuarios que hayan recibido equivocadamente los privilegios directamente sobre la tabla, en lugar de por medio del grupo.

```
 Role name | Attributes |  Member of
-----------+------------+-------------
 user3     |            | {}

    Name    |  Access privileges   | [...]
------------+----------------------+--------
 catalogue  | admin=arwdDxt/admin +| [...]
            | uci_users=rd/admin   |
            | user3=arwd/admin     |

user3@uci=> SELECT * FROM operations.catalogue;
 item_id | name | location | price | responsible
---------+------+----------+-------+-------------
(0 rows)
```

```
user3@uci=> INSERT INTO operations.catalogue (name, price)
user3@uci-> VALUES ('From user3',8.50);
ERROR:  new row violates row level security policy for "catalogue"
```

Conforme a lo que esperábamos, usuarios que tienen acceso por medio de privilegios otorgardos directamente tienen acceso a la tabla, pero, como resultado de no estar cubiertos por una norma, cualquiera selección devuelve un conjunto vacío. Al intentar añadir filas, por la misma razón, estos usuarios reciben un error de violación de normas.

El ejemplo que mostramos demuestra la eficacia del sistema de seguridad a nivel de filas. Una implementación del mismo concepto, utilizando técnicas clásicas requiere por lo menos una vista y una función disparadora para manejar los casos de `INSERT`, `UPDATE` y `DELETE`. Mientras más complejo se vuelvan los requisitos, más variaciones pueden ser necesarias. Claro que con la seguridad a nivel de filas también se tienen que implementar varias normas. La diferencia es que su gestión se ubica en el mismo contexto de los privilegios, es decir del sistema de autorización a nivel de tabla sin hacer falta generar objetos adicionales.

Las condiciones definidas para los comandos no tienen necesariamente porqué estar relacionadas directamente con campos en la tabla. Simplemente tienen que devolver un valor `true` o `false`. Supongamos que una empresa, cuyos ingresos se originan en la venta de cotizaciones de bolsa tenga dos tipos de clientes con diferentes ventanas temporales de acceso. Con los mecanismos de la seguridad a nivel de filas es muy sencillo definir una norma para cada grupo.

Clientes “always” con acceso 24x7 a los datos.

```
CREATE POLICY ON cotizaciones
FOR SELECT TO always
USING (true);
```

Clientes “working_days” con accesso de lunes a viernes entre 8:00 y 17:00.

```
CREATE POLICY working_days ON cotizaciones
FOR SELECT TO working_days
USING (current_time BETWEEN '08:00:00' AND '17:00:00'
       AND date_part('dow', current_timestamp) NOT IN (0,6));
```

De manera análoga se puede seguir diseñando otras normas. Encontrar el balance y la línea de demarcación entre lo que es responsabilidad de la aplicación y de la base de datos entra, con estas capacidades adicionales, en un nivel de complejidad más alto. Pero esto ya no es el tema de este documento.

7 Agradecimientos

La idea para este texto surgió de una invitación del comité organizador de PgCuba para dar un curso corto sobre el tema autorización en PostgreSQL. Durante la preparación de la documentación necesaria para el curso me di cuenta de que aunque presente, la información sobre este tema es distribuida en varias partes de la documentación oficial de PostgreSQL y en en un sinnúmero de contribuciones en internet. El tema autorización se encuentra también en manuales de administración, por lo general en un capítulo bastante corto y muy apegado a la forma de la documentación oficial. Lo que no encontré fue una presentación compacta de los problemas, soluciones y dependencias apta para proporcionar una visión de conjunto sobre la materia. Este era precisamente el objetivo del curso en La Habana en Octubre de 2015. Así que basicamente aproveché de esta oportunidad para poner por escrito los elementos del curso. Quiero agradecer en este lugar al comité organizador de PgCuba para su invitación y de forma muy particular a la MSc. Yudisney Vazquez Ortíz de la Universidad de Ciencias Informáticas de La Habana, Cuba, al MSc. Gilberto Castillo Martínez de ETECSA, Cuba y al Dr. Robert Meyer del Grupo Suizo de Usuarios de Linux, Zurich, Suiza por su lectura crítica del presente texto. La responsabilidad por cualquier inexactitud es, obviamente, unicamente del autor.

8 Anexos

8.1 Anexo 1: Lista de los privilegios

Fuente: [Pos15a], p. 1527.

La lista se encuentra en los comentarios sobre el comando SQL `GRANT` y muestra las letras utilizadas en las tablas de sistema y en los correspondientes comandos de línea en psql.

```
rolename=xxxx -- privileges granted to a role
=xxxx -- privileges granted to PUBLIC
r -- SELECT ("read")
w -- UPDATE ("write")
a -- INSERT ("append")
d -- DELETE
D -- TRUNCATE
x -- REFERENCES
t -- TRIGGER
X -- EXECUTE
U -- USAGE
C -- CREATE
c -- CONNECT
T -- TEMPORARY
arwdDxt -- ALL PRIVILEGES (for tables, varies for other objects)
* -- grant option for preceding privilege
/yyyy -- role that granted this privilege
```

Ejemplo:

```
swisspug@swisspug=> \dp
                                          Access privileges
   Schema   |         Name          |   Type   |        Access privileges        |
------------+-----------------------+----------+---------------------------------+
 operations | members               | table    | swisspug_operator=arwd/swisspug+|
            |                       |          | swisspug=arwdDxt/swisspug       |
 public     | v_member_list         | view     | swisspug_reader=r/swisspug     +|
            |                       |          | swisspug=arwdDxt/swisspug       |
```

Es la base de datos swisspug el administrador tiene el mismo nombre, es el proprietario de todos los objetos y no es superuser. Entre otros swisspug tiene dos esquemas y una tabla members sobre la que existe una vista para una aplicación web. La tabla members es

el contenedor de los datos y la vista hace una selección adecuada para visualizar la lista de los miembros en una pagina web. Como vemos el grupo swisspug_operator puede leer, insertar y modificar filas en la table members. El administrador es quien otorgó los privilegios. Usuarios en el grupo swisspug_operator pueden entonces gestionar la tabla haciendo todos los cambios necesarios como añadir nuevos miembros, cambios de dirección de miembros, etc. La vista por su naturaleza reflejará siempre el estado actual de la tabla según los criterios de selección implementados. Para proporcionar datos a la aplicación de web no queremos otorgar acceso directo a la tabla, ya que esta tiene muchas más informaciones que las que se necesitan para visualizar la lista de miembros en el sitio internet. Además no queremos que la aplicación web sea usada para alguien que pueda modificar los datos. Por esta razón hacemos la selección por medio de la vista y otorgaremos tan solo el privilegio `SELECT` al usuario swisspug_reader.

Bibliografía

[AFS15] Ibrar Ahmed, Asif Fayyaz, and Amjad Shahzad. *PostgreSQL Developer's Guide*. Packt Publishing, 1 edition, 2 2015.

[CKMP14] Paolo Corti, Thomas J. Kraft, Stephen V. Mather, and Bborie Park. *PostGIS Cookbook*. Packt Publishing, 1 edition, 2 2014.

[EH11] Peter Eisentraut and Bernd Helmle. *PostgreSQL Administration*. O'Really, 2 edition, 2011.

[KMR13] Hannu Krosing, Jim Mlodgenski, and Kirk Roybal. *PostgreSQL Server Programming*. Packt Publishing, 1 edition, 6 2013.

[OH15] Regina Obe and Leo S. Hsu. *PostGIS In Action*. Packt Publishing, 2 edition, 2015.

[Pos15a] PostgreSQL Global Development Group. *PostgreSQL 9.4.0 Documentation*. 2015. http://www.postgresql.org/files/documentation/pdf/9.4/postgresql-9.4-A4.pdf.

[Pos15b] PostgreSQL Global Development Group. *Row Security Policies*. 2015. http://www.postgresql.org/docs/9.5/static/ddl-rowsecurity.html.

[RKCB15] Simon Riggs, Hannu Krosing, Gianni Ciolli, and Gabriele Bartolini. *PostgreSQL 9 Administration Cookbook*. Packt Publishing, 2 edition, 4 2015.

[RN11] A. Rubinos and H. A. Nuevo. Seguridad en bases de datos. *Revista Cubana de Ciencias Informáticas*, 5(1), January-March 2011.

[Sch14] Hans-Jürgen Schönig. *PostgreSQL Administration Essentials*. Packt Publishing, 1 edition, 10 2014.

[Sch15] Hans-Jürgen Schönig. *Troubleshooting PostgreSQL*. Packt Publishing, 1 edition, 3 2015.

[SLVO15] Anthony R. Sotolongo León and Yudisney Vazquez Ortíz. *PL/pgSQL y otros lenguajes procedurales en PostgreSQL*. lulu.com, 1 edition, 3 2015. http://www.lulu.com.