75.15 / 75.28 / 95.05 - Base de Datos

Seguridad en Bases de Datos

Mariano Beiró

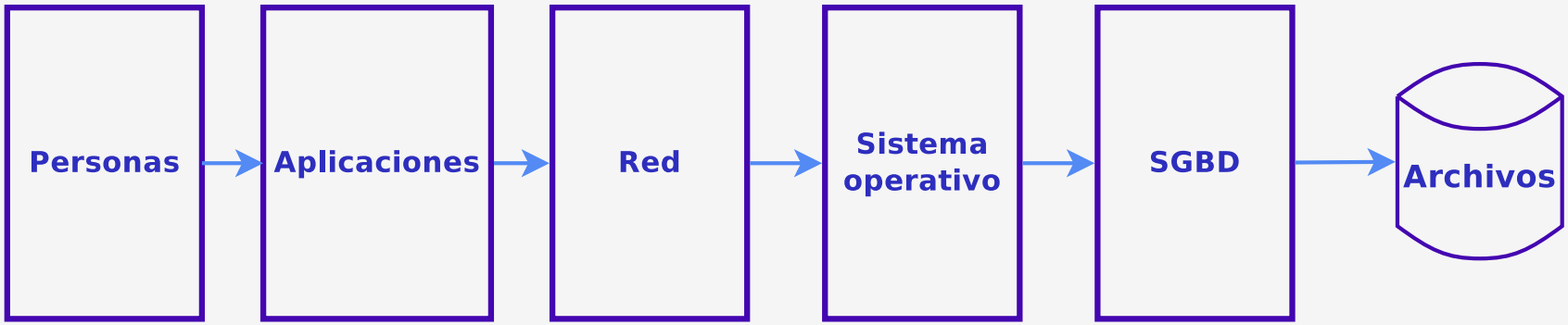
Dpto. de Computación - Facultad de Ingeniería (UBA)

14 de junio de 2022

Topics

Seguridad de la información

- La seguridad de la información es el conjunto de procedimientos y medidas tomadas para proteger a los componentes de los sistemas de información.
- Implica los siguientes factores:
  - Confidencialidad: No ofrecer la información a individuos no autoriz.
  - Integridad: Asegurar su correctitud durante el ciclo de vida.
  - Disponibilidad: Asegurar que la información esté disponible cuando es requerida por personas autorizadas.
  - No repudio: Que nadie que haya accedido a cierta información pueda negar haberlo hecho.
- Y debe ser cubierta en distintos niveles:



Políticas de seguridad de datos

- Las organizaciones utilizan distintas técnicas para resguardar la seguridad de datos:
  - Encriptado, firewalls, manejo de usuarios y permisos, comunicaciones seguras (ssh), etc.
- Aquí nos enfocaremos en la seguridad de datos, es decir, en la protección de los datos almacenados en un SGBD. Esta es sólo una de las facetas de la seguridad de la información.
- Toda organización debería contar con una política de seguridad de datos (data security policy).
- Ejemplo: Universidad de Berkeley
  - Data Security Policy: `https://berkeleycollege.edu/pdf/data-security-policy-statement.pdf`
  - Data best practices: [link]
  - How to protect against SQL injection: [link]

Historia

[ELM16 30.2 30.3]

- En la década del ’80 el Departamento de Defensa de los EE.UU. desarrolló dos sistemas de control de acceso a la información: DAC y MAC, con el objetivo de prevenir accesos no autorizados a información clasificada.

DAC (Discretionary Access Control)

Un objeto sólo puede ser accedido por el usuario o grupo al que pertenece. Existe el concepto de *dueño (owner)*. El dueño puede permitir el acceso a su recurso a otro usuario o grupo de usuarios. Ejemplo: sistema UNIX.

MAC (Mandatory Access Control)

El control de acceso es determinado por el sistema, quien establece una serie de relaciones entre los usuarios y los objetos (típicamente se asignan *niveles* a ambos). Históricamente ha sido utilizado en sistemas con múltiples jerarquías y que almacenan información altamente sensible. Ejemplo: organismos de defensa, milicias.

Control de acceso basado en roles (RBAC)

[ELM16 30.3.2]

- El control de acceso basado en roles *(RBAC, role based access control)* surgió en 1992 como una evolución de DAC, buscando adaptar los mecanismos de seguridad de la información a la estructura de las organizaciones, que cada día estaban más digitalizadas.
- Hoy en día RBAC es ampliamente utilizado por una mayoría de SGBD's y es parte del estándar SQL desde 1999.

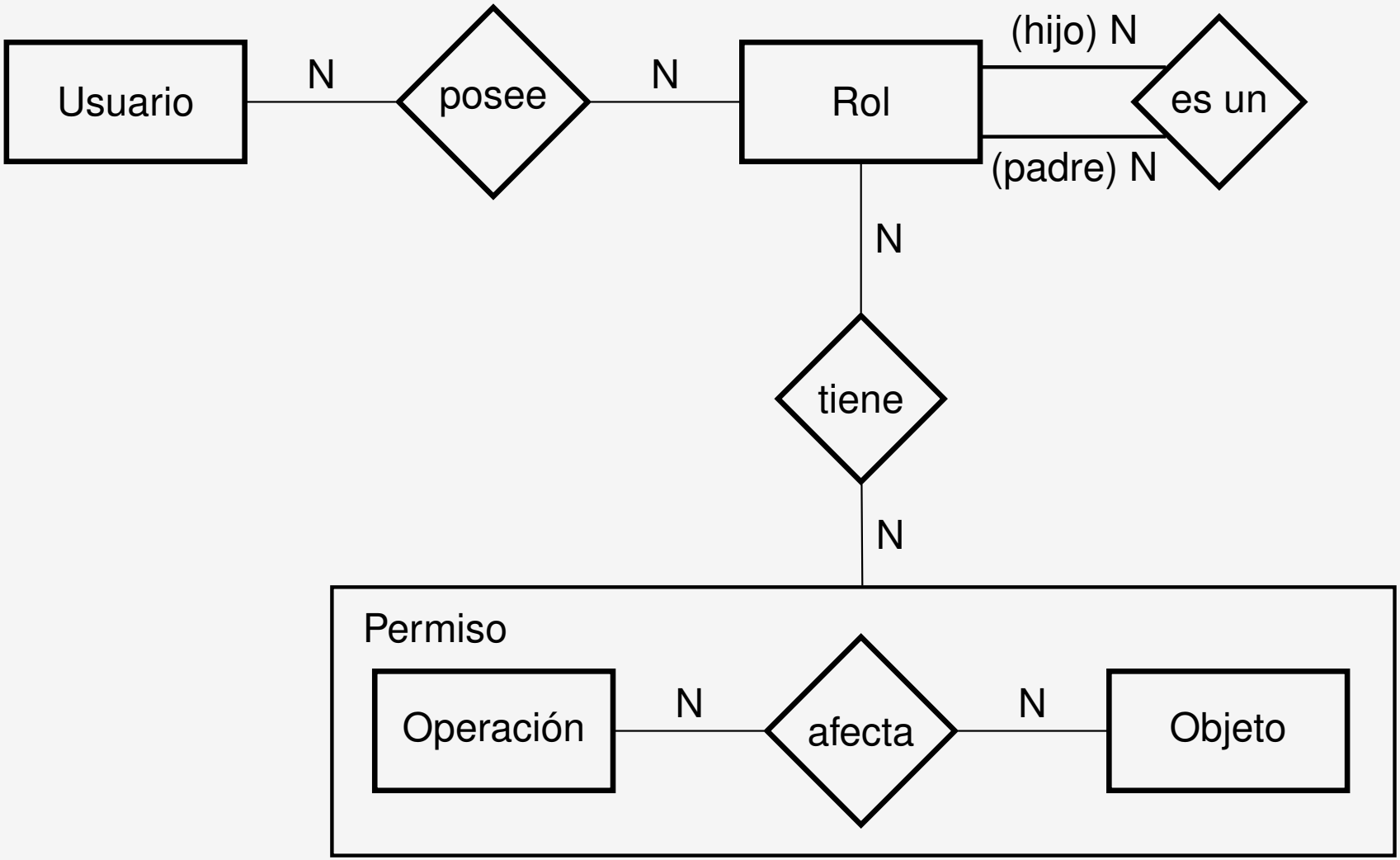
Control de acceso basado en roles (RBAC)

Componentes

- RBAC está basado en la definición de roles para las distintas actividades y funciones desarrolladas por los miembros de una organización, con el objetivo de regular el acceso de los usuarios a los recursos disponibles.
- Se utiliza la siguiente terminología:
  - Usuarios: Son las personas.
  - Roles: Son conjuntos de funciones y responsabilidades.
  - Objetos: Son aquello a ser protegido. En el caso de los SGBD's, son las tablas, esquemas, documentos, columnas, ...
  - Operaciones: Son las acciones que pueden realizarse sobre los objetos.
  - Permisos: Acciones concedidas o revocadas a un usuario o rol sobre un objeto determinado.

Control de acceso basado en roles (RBAC)

Modelo de datos de RBAC



Control de acceso basado en roles (RBAC)

Ejemplos

Este modelo permite expresar restricciones como:

- *"Esther Maio"* es la *"gerente de Ventas"*.
- Un *"gerente de Ventas"* es un *"empleado del Departamento de Ventas"*.
- Un *"empleado del Departamento de Ventas"* es *"empleado de la empresa"*.
- Un *"empleado de la empresa"* puede *"ver sus recibos de sueldo"*.
- Un *"empleado del Departamento de Ventas"* puede *"ingresar el detalle de una nueva venta"*.
- Un *"empleado del Departamento de Ventas"* puede *"ver el listado de clientes morosos"*.
- El *"gerente de Ventas"* puede *"generar reclamos a clientes morosos"*.

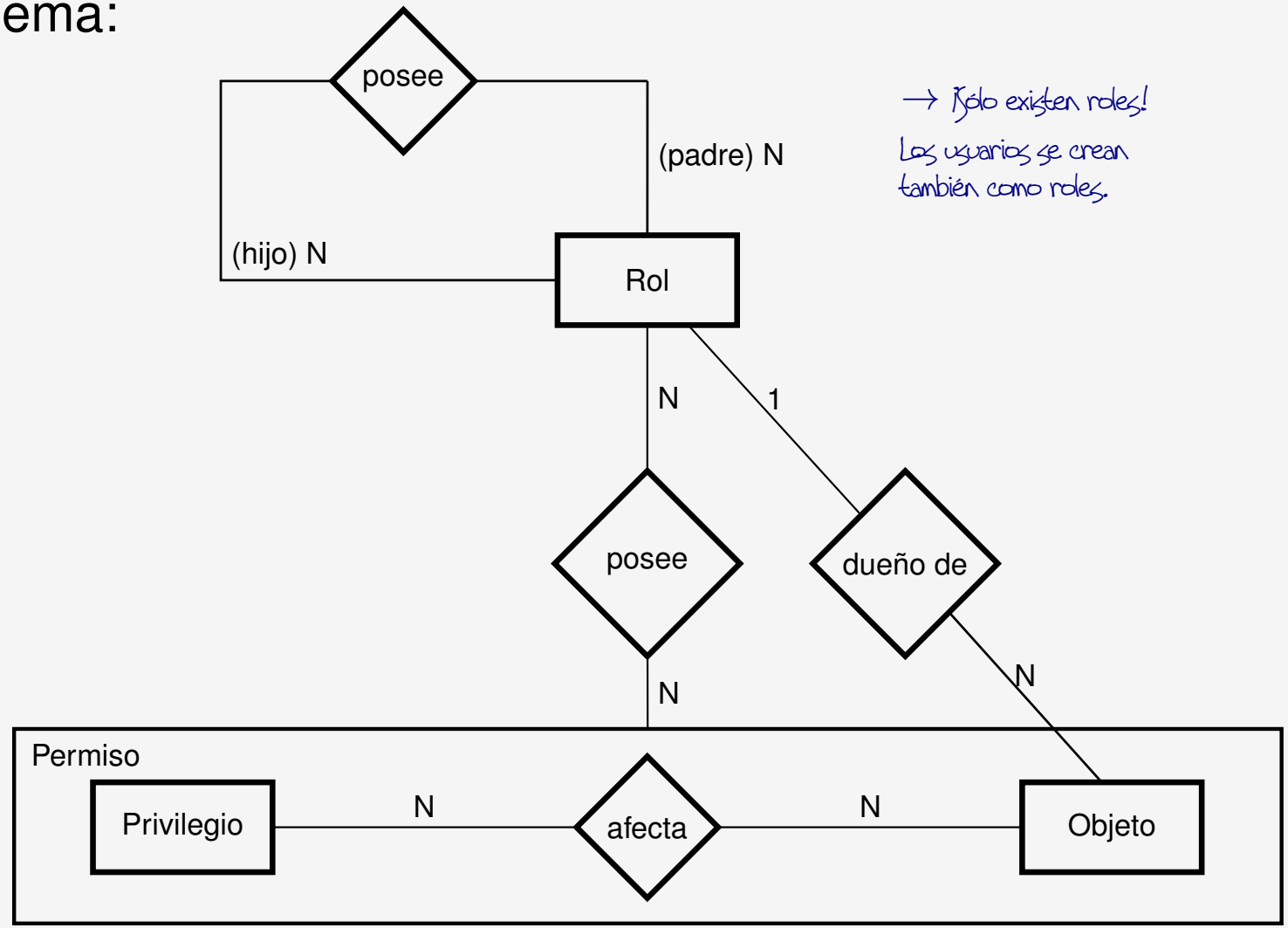
Control de acceso basado en roles (RBAC)

Características

- Existe una jerarquía de roles.
- Soporta la implementación de tres principios de seguridad:
  1. Criterio del *menor privilegio posible*: si un usuario no va a realizar una determinada operación, entonces no debe tener permisos para realizarla.
  2. División de responsabilidades: Se busca que nadie tenga suficientes privilegios para utilizar el sistema en beneficio propio. Para completar una tarea sensible debería ser necesaria la participación de roles mutuamente excluyentes, cada uno realizando una operación distinta.
     - Ejemplo: Que el pago de una factura a un proveedor requiera la aprobación de un empleado de Pagos y la carga de los datos de los proveedores deba realizarla un empleado de Compras.
  3. Abstracción de datos: Los permisos son abstractos. Las operaciones permitidas/prohibidas dependen de las propiedades del objeto en cuestión.

RBAC en Postgres

- La implementación de RBAC en Postgres sigue el siguiente esquema:



→ ¡Sólo existen roles!
Los usuarios se crean también como roles.

Tablas, esquemas y bases

- En postgres la cima de la jerarquía de objetos son las bases *(databases)*.
- Una base está conformada por cero o más esquemas *(schemas)*, y un esquema es un conjunto de tablas (aunque también contiene otros objetos, como funciones, vistas, tipos de dato, etc.).

Nota

La diferencia entre bases y esquemas es que una conexión al servidor postgres debe realizarse a una única base, aunque puede trabajar con más de un esquema de dicha base. Los esquemas son una separación lógica de las tablas, mientras que las bases están físicamente separadas.
Cuando instalamos postgres, automáticamente se configura una base `postgres` con un esquema `public`.

Gestión de usuarios y bases en Postgres

- Después de la instalación únicamente existe el usuario `postgres`, y una base `postgres` con un esquema `public`.
- Postgres cuenta con una serie de comandos para trabajar:
  - `createuser` (crear roles)
  - `dropuser` (eliminar roles)
  - `createdb` (crear bases)
  - `dropdb` (eliminar bases)
  - `psql` (**cliente** de postgres)
  - `pg_dump` (importación de datos externos)

Modos de autenticación

Para ejecutar un comando con un usuario X debemos estar logueados con ese mismo usuario en el sistema operativo.
Este criterio puede modificarse cambiando la autenticación de "peer" a "password" u otro tipo en el archivo `pg_hba.conf` (https://www.postgresql.org/docs/current/static/auth-methods.html)

Gestión de usuarios y bases en Postgres

Creación de usuarios

- La creación de usuarios, grupos y roles se realiza con el comando **CREATE ROLE**. Su sintaxis completa depende de cada SGBD.
- En Postgres tenemos:

```
# CREATE ROLE nombre_usuario { SUPERUSER | CREATEDB |
                | CREATEROLE | CREATEUSER | LOGIN |
                | PASSWORD clave | IN ROLE rol_1, rol_2, ... | ... };
```

- La opción **SUPERUSER** indica que el usuario será superusuario, es decir que podrá sobrepasar todas las restricciones de acceso.
- La opción **IN ROLE** asigna inmediatamente al nuevo usuario roles ya existentes.
- La opción **LOGIN** es necesaria para que el rol pueda loguearse.

Gestión de usuarios y bases en Postgres

Creación de usuarios

- Nos conectamos con el usuario `postgres`...
- Y comenzaremos creando un rol de administrador que sea superusuario:

```
# CREATE ROLE admin LOGIN SUPERUSER;
```

- Y luego crearemos un usuario “lolo”:

```
# CREATE ROLE lolo LOGIN PASSWORD '111';
```

- También podíamos haber hecho todo ésto con el comando `createuser`:

```
(      ) $  sudo su - postgres
postgres $  createuser -s admin        # Crea a admin como superusuario
postgres $  createuser lolo            # Crea al usuario lolo
```

Gestión de usuarios y bases en Postgres

Creación de una base de datos

- El comando **CREATE DATABASE** permite crear una nueva base en el servidor. Su sintaxis es:

```
# CREATE DATABASE nombre_db [ OWNER usuario ];
```

- La opción **OWNER** nos indica quién será el dueño de la base. Si no se indica, será el usuario que ejecuta el comando de creación.
- El usuario que ejecuta el comando debe tener permisos de **CREATE DATABASE**.
- También podemos usar el comando `createdb` desde la terminal del sistema operativo:

```
(      ) $  sudo su - postgres
postgres $  createdb -O admin nombre_esquema # Crea la base y le asigna
                                             # al usuario admin como owner
```

Gestión de usuarios y bases en Postgres

Creación de un esquema

- Dentro de una base, podemos crear un nuevo esquema con el comando **CREATE SCHEMA**. Su sintaxis (SQL estándar) es:

```
# CREATE SCHEMA nombre_esquema [ AUTHENTICATION usuario ];
```

- La opción **AUTHENTICATION** nos indica quién será el dueño del esquema. Si no se indica, será el usuario que ejecuta el comando de creación.

Gestión de usuarios y bases en Postgres

Comandos del entorno **psql**

- Cuando trabajamos dentro de psql, el prompt nos indica si el usuario es:

```
db_empresa=>        # usuario normal
db_empresa=#        # superusuario
```

- `\l` muestra el listado de bases de datos y quién es el dueño de cada una.
- `\dn` muestra el listado de esquemas de la base de datos a la que estamos conectados, y quién es el dueño de cada uno.
- `\dt` muestra el listado de tablas de todos los esquemas de la base de datos a la que estamos conectados, y quién es el dueño de cada una.
- `\dg` muestra los roles existentes, con sus atributos y jerarquías.
- `\dp` muestra los privilegios que poseen los usuarios sobre los objetos.
- Con `TAB` autocompleta/sugiere.

Permisos en Postgres

Privilegios

- Recordemos que en Postgres los permisos vinculan roles con privilegios y objetos.
- Los principales tipos de privilegio en Postgres son:
  - `INSERT [ lista_columnas ]`
  - `UPDATE [ lista_columnas ]`
  - `DELETE`
  - `SELECT [ lista_columnas ]`
  - `REFERENCES [ lista_columnas ]`
  - `USAGE`
  - `CREATE`
  - `EXECUTE`
  - `ALL PRIVILEGES`

Permisos en Postgres

Objetos y dueños

- Mientras que los principales tipos de objeto son:
  - `DATABASE`
  - `TABLE`
  - `VIEW`
  - `INDEX`
  - `SCHEMA`
  - `FUNCTION`
- El dueño de un objeto tiene todos los privilegios sobre el objeto, y puede extenderlos a los usuarios que desee a través del comando `GRANT` de SQL. Inclusive puede asignar a otro rol como dueño, y dejar de serlo.
- ¡Atención! Los superusuarios tienen todos los privilegios sobre todos los objetos, y son irrevocables. (En realidad ni siquiera están sometidos al control de acceso). También pueden conceder cualquier tipo de privilegio a cualquier usuario sobre cualquier objeto.

Permisos en Postgres

GRANT's y REVOKE's [ELM16 30.2]

- Los privilegios se conceden con el comando **GRANT** y se revocan con el comando **REVOKE**.
- Su sintaxis básica (SQL estándar) es:

```
GRANT {privilegio}
ON {[ TABLE | DATABASE | SCHEMA | ... ] objeto}
TO {rol} [ WITH GRANT OPTION ];

REVOKE privilegio ON objeto FROM rol [ CASCADE | RESTRICT ];
```

- La opción **WITH GRANT OPTION** permite que el usuario que recibe los privilegios pueda a su vez extenderlos hacia otros usuarios.
- La opción **CASCADE** indica que si el usuario había extendido sus privilegios hacia otros, a estos últimos también se les quiten.
- Ejemplo:

```
GRANT INSERT ON Clientes TO gerente_ventas;
```

Permisos en Postgres

GRANT's y REVOKE's: Condiciones de concesión

- Para que el usuario U pueda conceder un privilegio P sobre un objeto O a un rol R, es necesario que se cumpla al menos una de las siguientes 3 condiciones:
  - Que U sea el dueño de O.
  - Que U disponga del privilegio P sobre el objeto O, concedido por algún rol R' *con permiso de extenderlo* (`WITH GRANT OPTION`).
  - Que U sea superusuario.

Nota

Para rastrear y documentar el origen de los privilegios sobre un objeto dado es útil construir un grafo de concesiones *(grants)*. En este grafo se ubica a los roles como nodos, y se agrega un eje dirigido (u, v) cuando el usuario u haya concedido privilegios sobre dicho objeto a v.

Permisos en Postgres

Permisos a nivel de fila

- Si bien no son estándar en SQL, Postgres también permite configurar políticas a nivel de fila utilizando el comando **CREATE POLICY**.
- Ejemplo: Cada vendedor de la empresa tiene un grupo de clientes a cargo, identificados porque el atributo 'vendedor' de la tabla *Clientes* contiene como valor el nombre de usuario del vendedor asignado.

```
ALTER TABLE Clientes ENABLE ROW LEVEL SECURITY;
CREATE POLICY mi_pol ON Clientes FOR ALL USING (vendedor = current_user);
```

- Cuando el vendedor haga una consulta, únicamente podrá ver a sus propios clientes.

Permisos en Postgres

Limitaciones

- Hay cosas que no podemos cambiar...

- Cuando creamos una tabla, el dueño inicial será quien la crea.
- Un `DROP TABLE` ó un `ALTER TABLE` sólo puede ser ejecutado por el dueño de la tabla o por un superusuario.
- ... pero podemos cambiar la asignación del dueño si lo hace el dueño o un superusuario:
  - `ALTER TABLE OWNER TO juan`

- El dueño de una base de datos es quien la crea.
- Y él es el único que puede ejecutar un `DROP DATABASE`.
- ... aunque también se puede cambiar la asignación del dueño de la base de datos:
  - `ALTER DATABASE OWNER TO juan`

- Idem para los esquemas (`ALTER SCHEMA`).

- Conclusión: Los permisos de `DROP` y `ALTER` no son *concedibles*.

- Sólo los superusuarios pueden crear nuevos superusuarios.

Permisos en Postgres

Los esquemas `public`

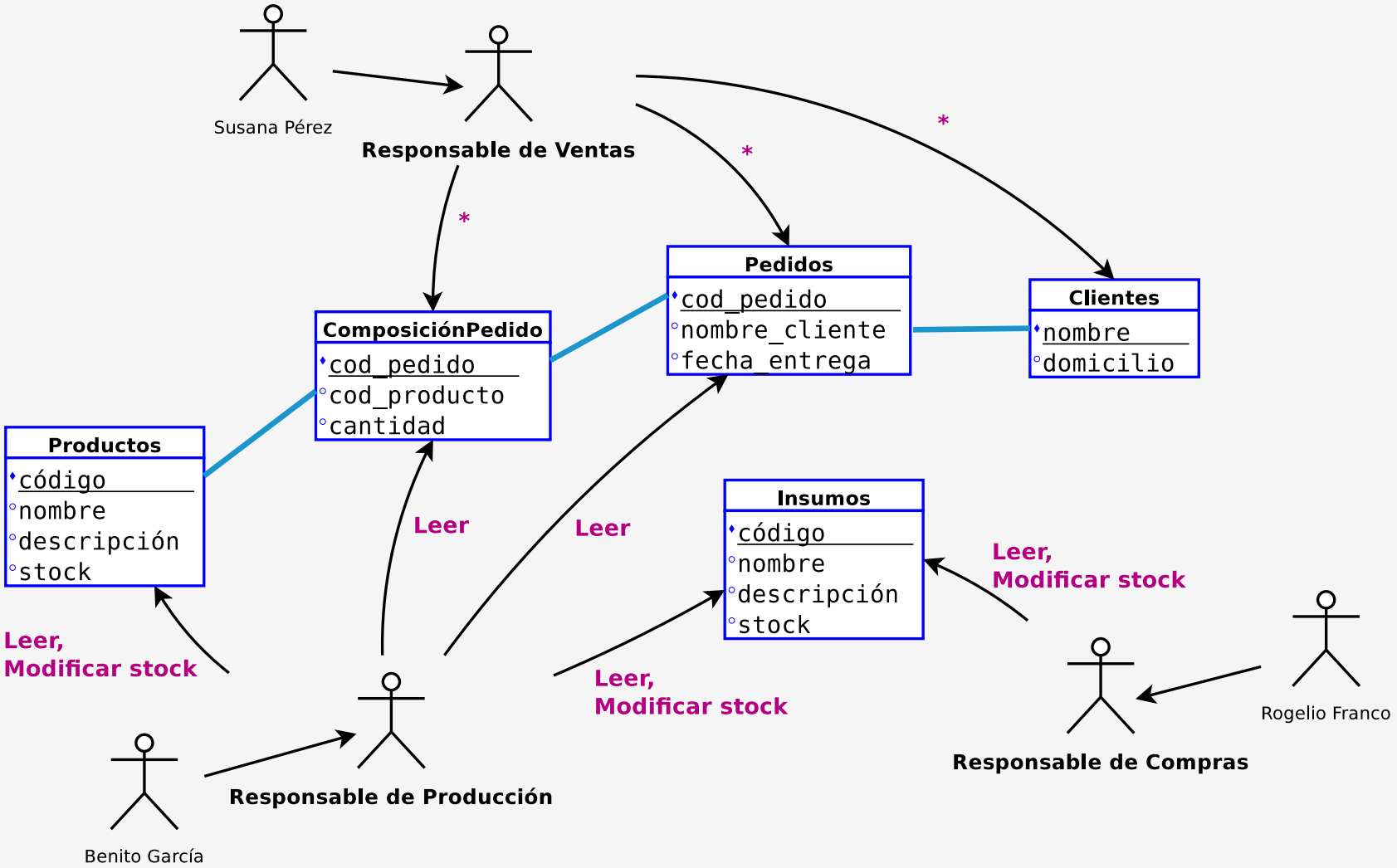
- La base `postgres` de la que disponemos inicialmente contiene un único esquema: `public`.
- Cuando creamos una nueva base con `CREATE DATABASE` se creará un esquema `public` automáticamente.
- Los esquemas `public`:
  - No obligan a especificar su nombre cuando nos referimos a una tabla dentro de ellos (cómodo).
  - Pero son demasiado permisivos: no chequean que hayan permisos de `CREATE` cuando creamos una tabla en ellos.

Recomendación

Para evitar problemas de seguridad, generalmente se recomienda no trabajar con los esquemas `public` y eliminarlos desde un comienzo.
Es preferible crear nuestro propio esquema y conceder permisos de `USAGE` a los usuarios.

Ejemplo

- Consideremos la siguiente situación:



Ejemplo

- Comenzaremos por crear los usuarios con su jerarquía de roles:

```
CREATE ROLE resp_ventas;
CREATE ROLE resp_compras;
CREATE ROLE resp_produccion;
CREATE ROLE sperez LOGIN PASSWORD='susanita' IN ROLE resp_ventas;
CREATE ROLE rfranco LOGIN PASSWORD='roger' IN ROLE resp_compras;
CREATE ROLE bgarcia LOGIN PASSWORD='1234' IN ROLE resp_produccion;
```

Ejemplo

- Ahora creamos el esquema y las tablas que usaremos:

```
DROP SCHEMA public;
CREATE SCHEMA empresa;
GRANT USAGE ON SCHEMA empresa TO ALL;
SET search_path to empresa;

CREATE TABLE Clientes(nombre VARCHAR(20) PRIMARY KEY, domicilio VARCHAR(30));
CREATE TABLE Pedidos(cod_pedido INT PRIMARY KEY,
                     nombre_cliente VARCHAR(20) REFERENCES
                     Clientes(nombre), fecha_entrega DATE);
CREATE TABLE Productos(codigo INT PRIMARY KEY, nombre VARCHAR(20),
                       descripcion VARCHAR(50), stock INT);
CREATE TABLE Insumos(codigo INT PRIMARY KEY, nombre VARCHAR(40),
                     descripcion VARCHAR(50), stock INT);
CREATE TABLE ComposicionPedido(cod_pedido INT REFERENCES Pedidos
              (cod_pedido), cod_producto INT REFERENCES Productos
              (codigo), cantidad INT, PRIMARY KEY (cod_pedido, cod_producto));
```

Ejemplo

- Y cargaremos algunos datos:

```
INSERT INTO Clientes VALUES ('La Sierra S.A.', 'Los Andes 280'),
                            ('Turdera S.R.L.', 'Ju rez 1539');

INSERT INTO Pedidos VALUES (58, 'La Sierra S.A.', '26-10-2017'),
                           (60, 'Turdera S.R.L.', '28-10-2017');

INSERT INTO Productos VALUES (381, 'Hamaca paraguaya', '...', 5),
                             (408, 'Sombrilla veneciana', '...', 3);

INSERT INTO Insumos VALUES (33, 'Rollo de Soga 200m, 4mm', '...', 15),
                           (38, 'Rollo de lona azul 100m', '...', 3);

INSERT INTO ComposicionPedido VALUES (58, 381, 1),
                                     (58, 408, 1),
                                     (60, 381, 2),
                                     (60, 408, 3);
```

Ejemplo

- A cada responsable le asignaremos los permisos necesarios para trabajar con las tablas que necesite:

```
GRANT ALL PRIVILEGES ON Clientes, Pedidos, ComposicionPedido TO resp_ventas;
GRANT SELECT, UPDATE (stock) ON Insumos, Productos TO resp_produccion;
GRANT SELECT ON Pedidos, ComposicionPedido TO resp_produccion;
GRANT SELECT, UPDATE (stock) ON Insumos TO resp_compras;
```

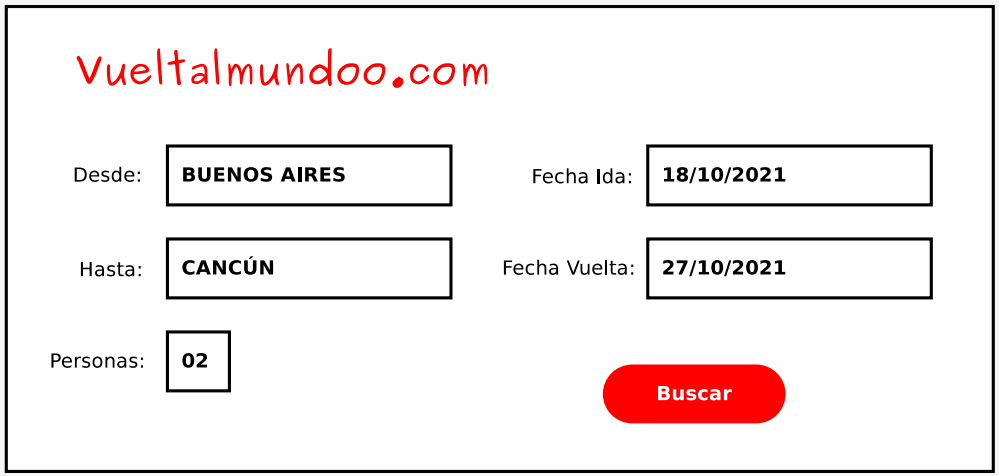
SQL Injection

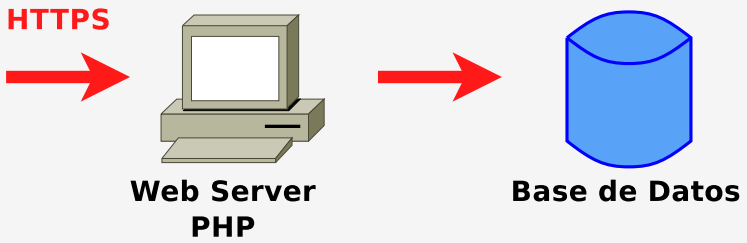
[ELM16 30.4.1]

- Los ataques de SQL Injection son una de las fallas de seguridad más frecuentes en los sistemas que emplean bases de datos relacionales.
- Consisten en manipular a nivel de aplicación las variables de entrada de una consulta SQL de manera de modificarla.
- Estos ataques pueden ocasionar graves consecuencias:
  - Acceso no autorizado a los datos
  - Eliminación malintencionada de datos (o hasta de la base de datos completa)
  - Alteración de datos o falsificación de información

Ejemplo

Conexión a una base de datos desde PHP



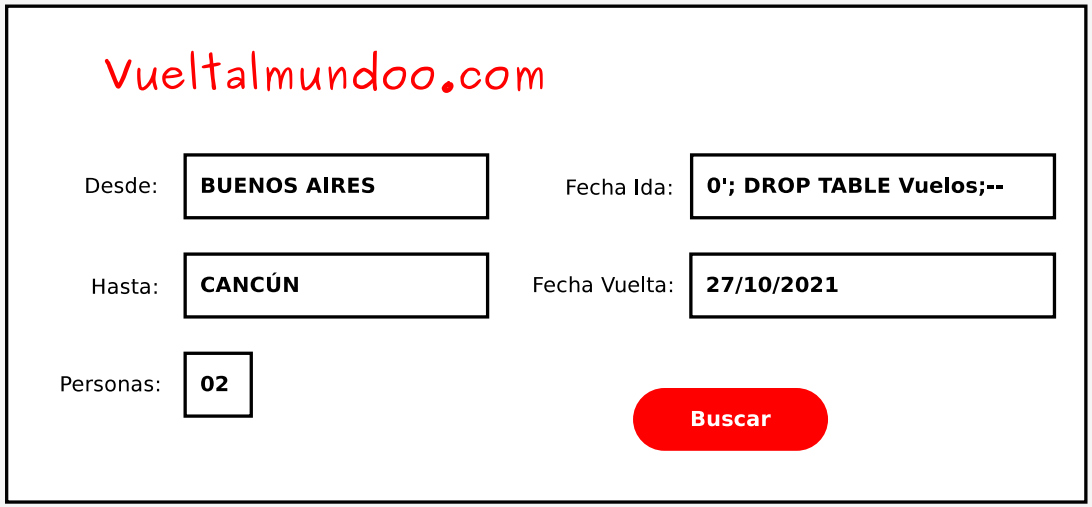


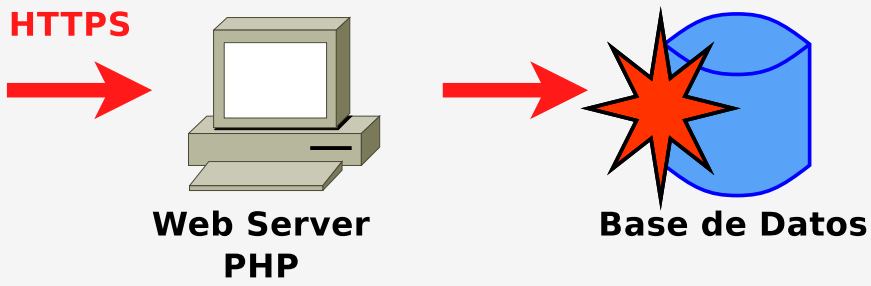

```
$lugarDesde = $_POST['desde'];
$lugarHasta = $_POST['hasta'];
$fechaIda = $_POST['ida'];
$consulta = "SELECT * FROM Vuelos
             WHERE desde='$lugarDesde' AND hasta='$lugarHasta'
             AND ida='$fechaIda'";
$resultado = $pdo->query($consulta);
...
```

Ejemplo

Conexión a una base de datos desde PHP

- Pero qué pasa si...






```
SELECT * FROM Vuelos
WHERE desde='Buenos Aires' AND hasta='Cancún'
AND ida='0'; DROP TABLE Vuelos; --'
```

Precauciones

```
$nombreUsuario = $_POST['usuario'];
$consulta = "SELECT domicilio FROM Usuarios WHERE nombre='$nombreUsuario'";
$resultado = $pdo->query($consulta);
...
```

1. Utilizar una función de escapeo sobre los parámetros antes de concatenarlos con el *string* de la consulta.
   - Ejemplo: mysql_real_escape() en PHP (Ya algo obsoleto). No es lo más conveniente, porque las reglas de escapeo pueden variar de un motor a otro.
2. Aprovechar los mecanismos de control de acceso (RBAC)
   - No dar privilegios de escritura si no es necesario! → (menor privilegio posible)
3. Validar los parámetros
   - Castear cada dato al tipo correspondiente antes de anexarlo a la consulta.
   - Validar que los valores del parámetro estén dentro de los que el usuario podía asignarle.

Precauciones

4. Utilizar consultas parametrizadas *(prepared statements)*.
   - Primero se define la estructura de la consulta, que el SGBD ya aprovecha para elaborar un plan de consulta eficiente.
   - Cuando se insertan los parámetros, el lenguaje se ocupa de hacer las conversiones necesarias.
   - Por ejemplo, en PHP (con PDO):

```
$consulta = $pdo->prepare("SELECT domicilio FROM Usuarios
                           WHERE nombre = :name");
$consulta->execute(array('name' => $nombreUsuario));
...
```

   - En Java (JDBC API):

```
PreparedStatement consulta = conn.prepareStatement("SELECT domicilio
                             FROM Usuarios WHERE nombre = ?");
consulta.setString(1, nombreUsuario);
consulta.executeQuery();
...
```

Bibliografía

[GM09] Database Systems, The Complete Book, 2nd Edition.
H. García-Molina, J. Ullman, J. Widom, 2009.
Capítulo 10.1

[ELM16] Fundamentals of Database Systems, 7th Edition.
R. Elmasri, S. Navathe, 2016.
Capítulo 30