

Almacenamiento de Datos Postgres

Instalación de PostgreSQL

En la Instalación de Odoo 8 en Windows wl motor de la BD PostgreSQL,la Base de datos Odoo, asi como su gestor pgadmin III se instala automáticamente. Pero no será este el gestor que usemos,Usaremos Navicat.

1 - Objetivos del tutorial de PostgreSQL.

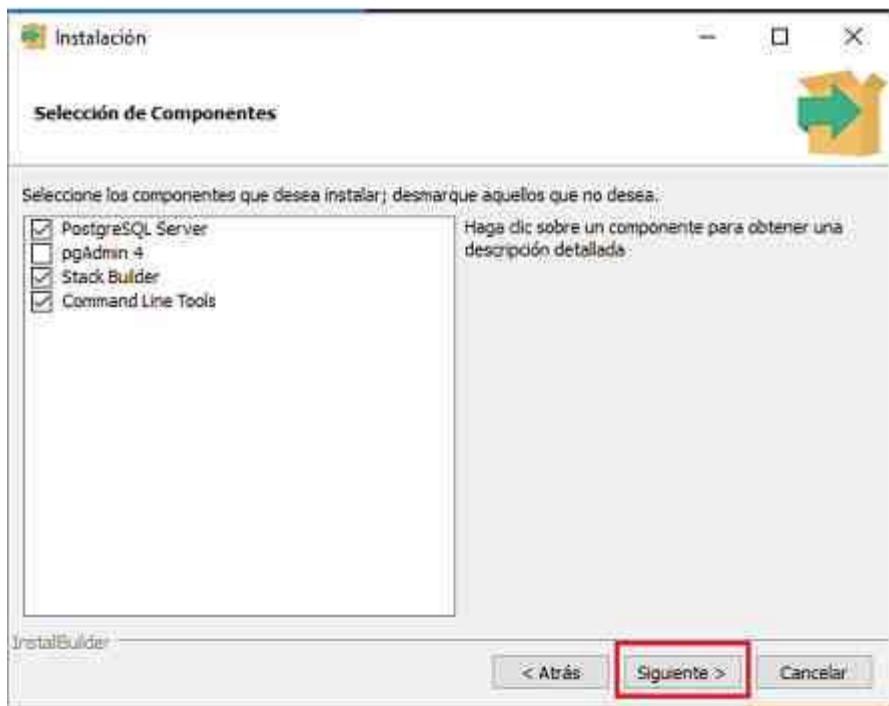
Todas las BD y tablas que necesita Odoo para trabajar están hechas en Postgresql, es necesario tener conocimiento de como funciona este SGDB relacional para comprender como Odoo almacena los datos

Esta parte del manual brinda un concepto teórico corto, luego un problema resuelto que invito a ejecutar, modificar y jugar con el mismo. Por último, y lo más importante, una serie de ejercicios propuestos que nos permitirá saber si podemos aplicar el concepto propuesto.

El primer paso será descargar el PostgreSQL.

El instalador nos guiará por una serie de pasos hasta tener ejecutando el servidor en forma local en nuestra computadora:

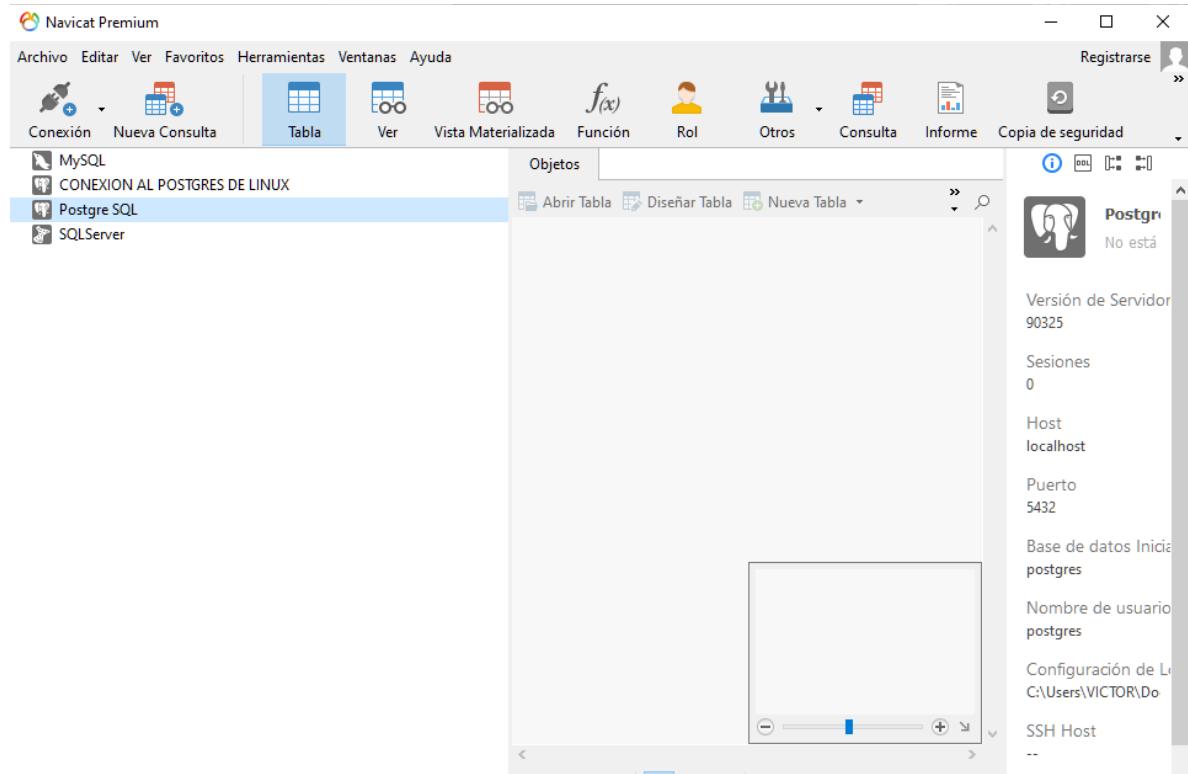




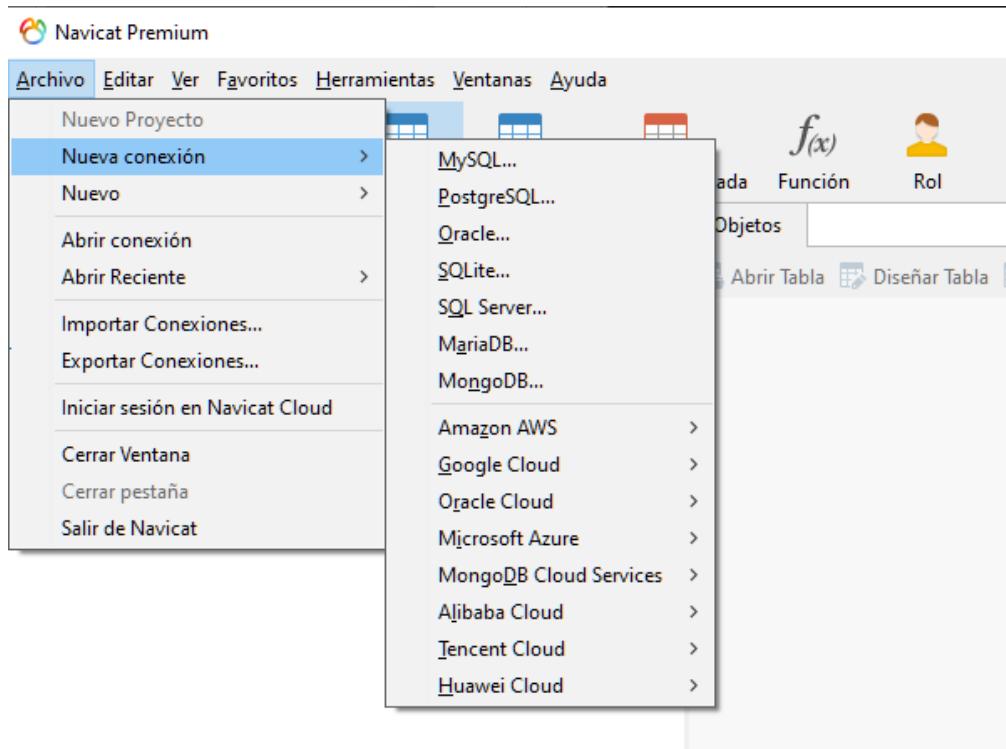
Al final se ejecuta el programa "Stack Builder" que nos permite instalar otros complementos para PostgreSQL, por el momento no instalaremos nada más (podemos cerrar el programa)

Junto con el servidor de base de datos PostgreSQL que acabamos de instalar necesitaríamos instalar un gestor de bases de datos llamado Navicat

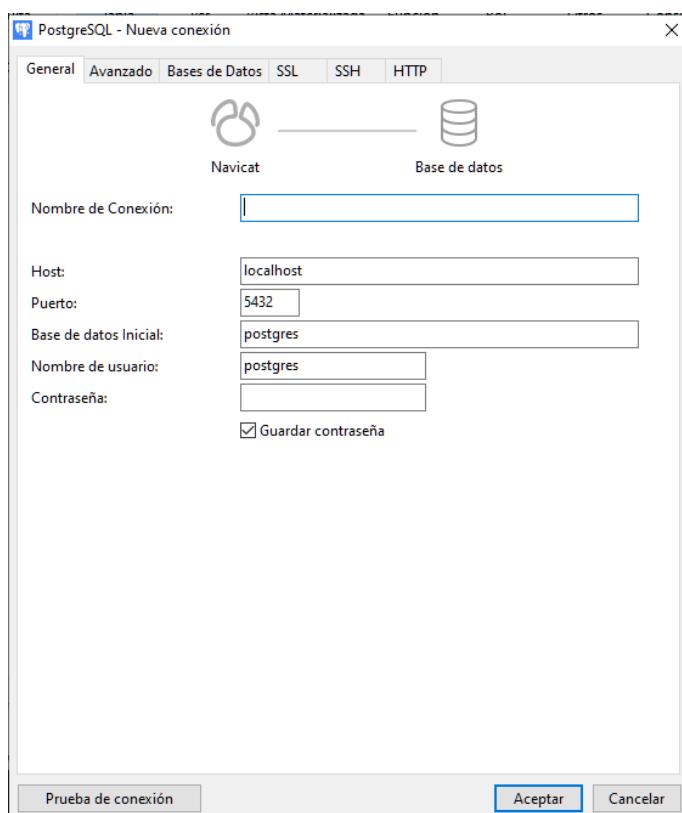
Ejecutemos el programa Navicat:



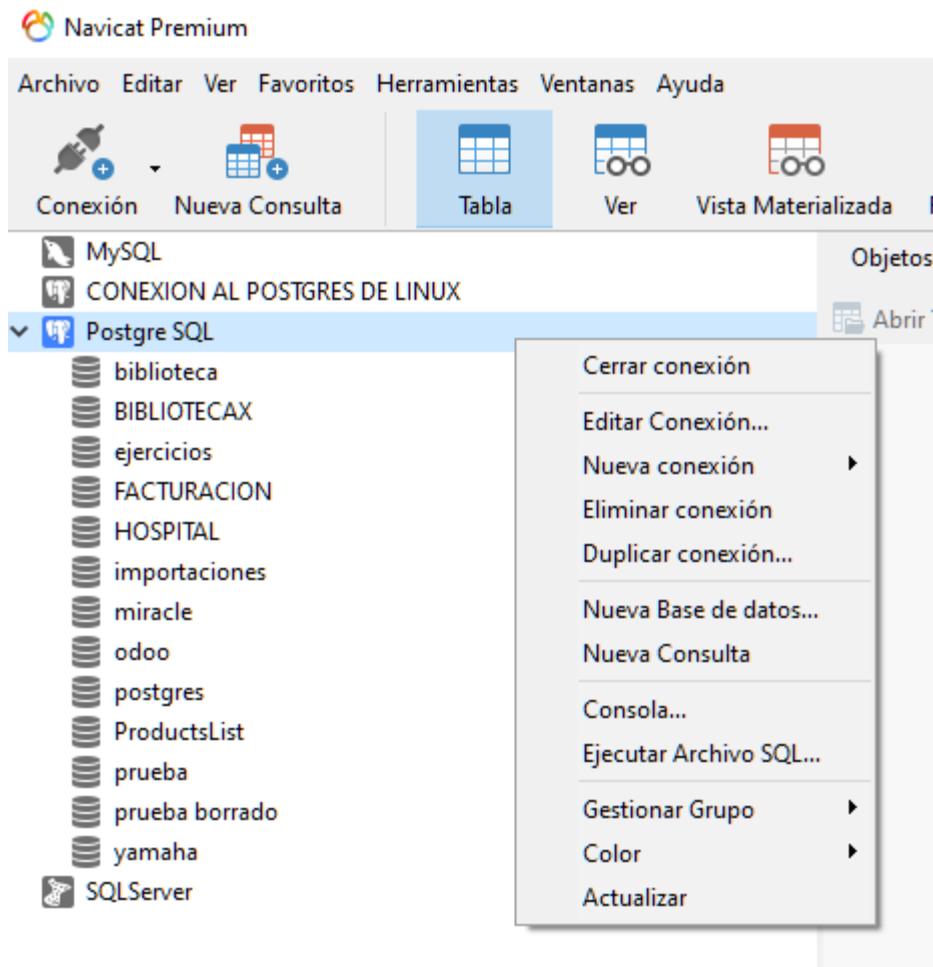
El programa navicat necesita crear una conexión a nuestro servidor postgresql



Nos solicita los datos de la conexión

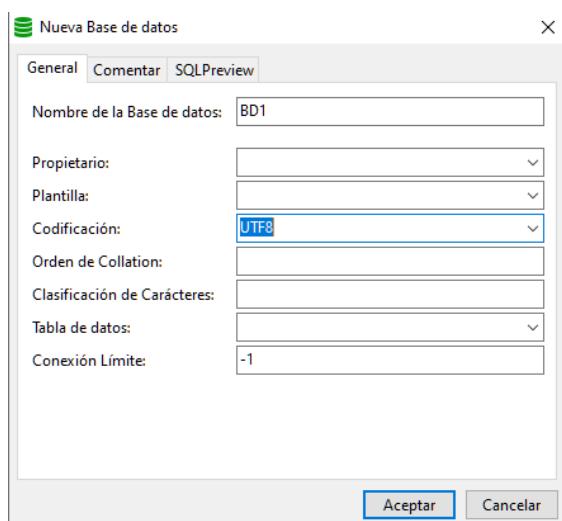


Una vez creada la conexión a nuestro servidor postgresql, y probada la conexión nos conectamos a el

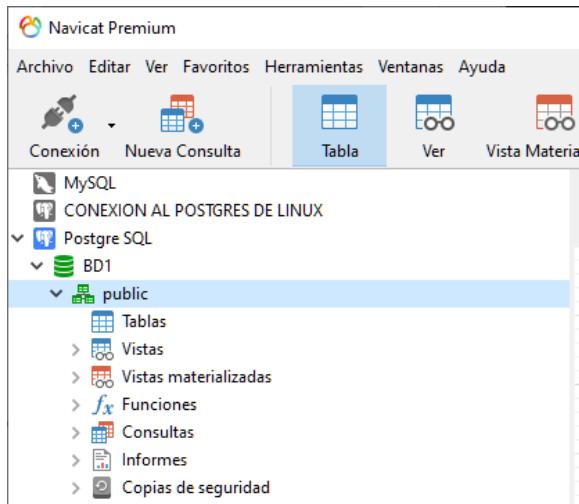


Para crear una base de datos presionamos el botón derecho del mouse donde dice nueva base de datos

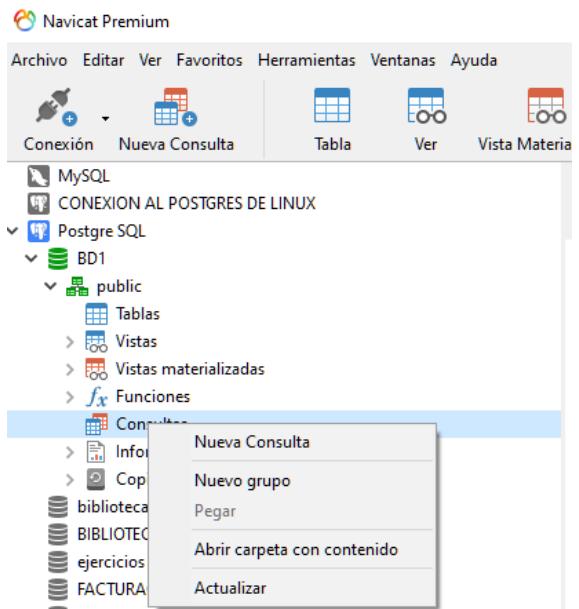
Aparece un diálogo donde debemos ingresar el nombre de la base de datos a crear, la llamaremos "BD1" y con codificación UTF-8:



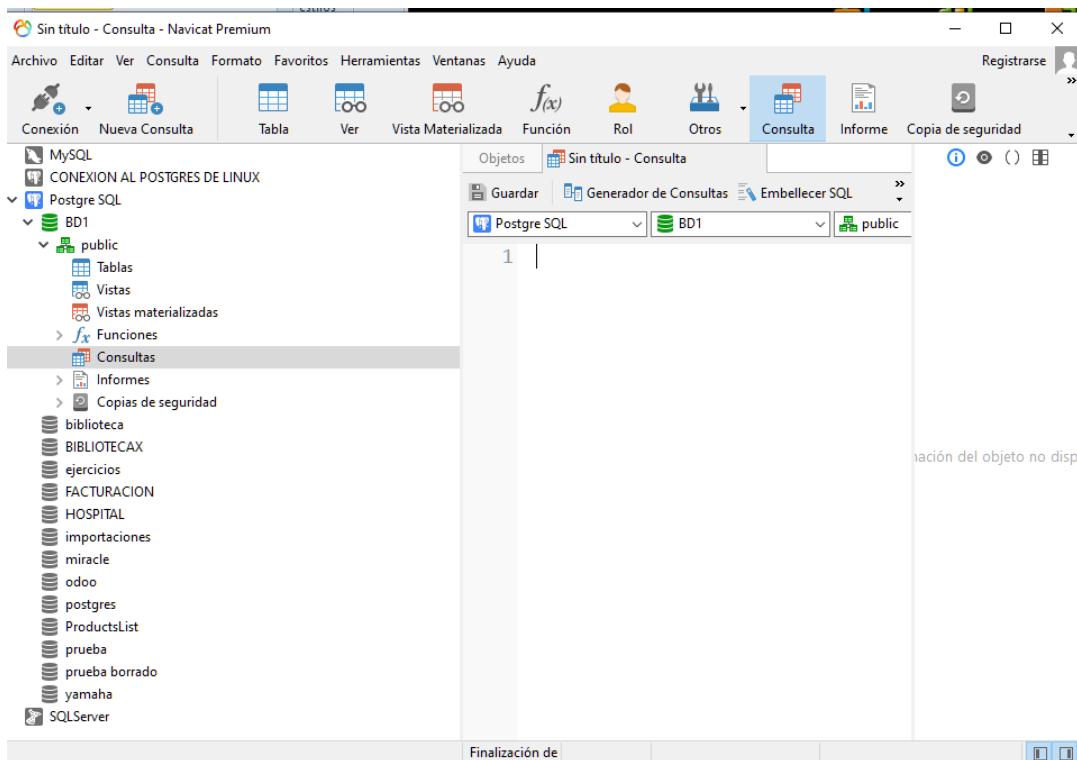
Ahora podemos seleccionar en la ventana de la izquierda la base de datos "bd1" que acabamos de crear:



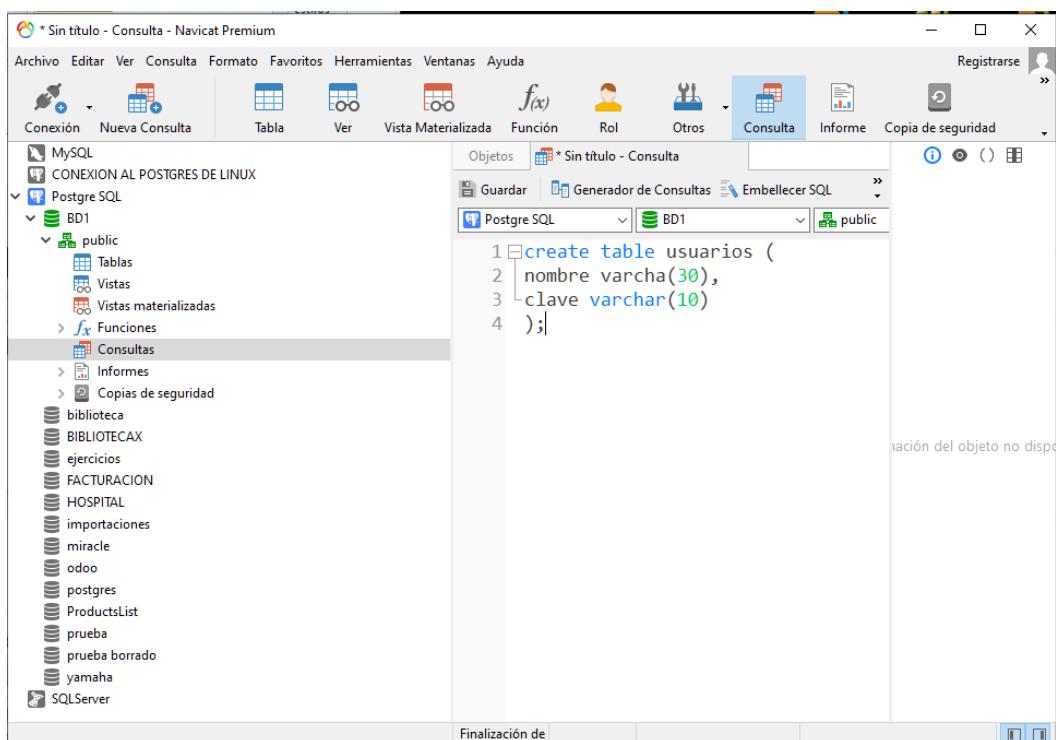
Para poder ejecutar comandos SQL debemos de elegir bajo consultas Nueva consulta



En los próximos conceptos que veamos escribiremos los comandos SQL dentro de la ventana de la consulta que acabamos de crear:



Probemos por ejemplo de ejecutar un comando SQL que crea una tabla llamada "usuarios" en la base de datos "bd1" (tema que veremos en el próximo concepto):



Presionamos el ícono de "Ejecutar" para que se ejecute el comando SQL "create table":

The screenshot shows the Navicat Premium interface for PostgreSQL. In the left sidebar, under the 'BD1' database, the 'public' schema is selected. In the main area, a SQL query window titled 'Sin título - Consulta' contains the following code:

```

1 create table usuarios (
2     nombre varchar(30),
3     clave varchar(10)
4 );

```

Si es correcto el comando SQL de "create table" se nos informa que la misma fue creada, la podemos ver abajo

The screenshot shows the same Navicat Premium interface after executing the SQL code. A message box titled 'Mensaje' displays the results:

```

create table usuarios (
    nombre varchar(30),
    clave varchar(10)
)
> OK
> Hora: 0,006s

```

The screenshot shows the object browser on the left side of the Navicat Premium interface. Under the 'BD1' database and 'public' schema, the 'Tables' section is expanded, showing a list of tables. The 'usuarios' table is highlighted, indicating it has been recently created.

En el apartado tablas ahora aparecerá la tabla recién creada

Si no aparece debemos presionar el botón derecho del mouse sobre "Tables" y seleccionar "Refresh":

Tipos de sentencias SQL y componentes sintácticos

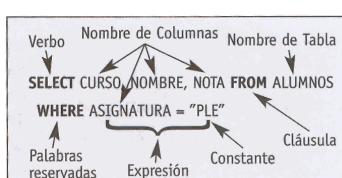
En SQL tenemos bastantes sentencias que se pueden utilizar para realizar diversas tareas. Dependiendo de las tareas, estas sentencias se pueden clasificar en tres grupos principales (DML, DDL, DCL), aunque nos quedaría otro grupo que a mi entender no está dentro del lenguaje SQL sino del PLSQL.

SENTENCIA		DESCRIPCIÓN
DML	Manipulación de datos SELECT INSERT DELETE UPDATE	Recupera datos de la base de datos. Añade nuevas filas de datos a la base de datos. Suprime filas de datos de la base de datos. Modifica datos existentes en la base de datos.
DDL	Definición de datos CREATE TABLE DROP TABLE ALTER TABLE CREATE VIEW DROP VIEW CREATE INDEX DROP INDEX CREATE SYNOYM DROP SYNONYM	Añade una nueva tabla a la base de datos. Suprime una tabla de la base de datos. Modifica la estructura de una tabla existente. Añade una nueva vista a la base de datos. Suprime una vista de la base de datos. Construye un índice para una columna. Suprime el índice para una columna. Define un alias para un nombre de tabla. Suprime un alias para un nombre de tabla.
DCL	Control de acceso GRANT REVOKE Control de transacciones COMMIT ROLLBACK	Concede privilegios de acceso a usuarios. Suprime privilegios de acceso a usuarios Finaliza la transacción actual. Aborata la transacción actual.
PLSQL	SQL Programático DECLARE OPEN FETCH CLOSE	Define un cursor para una consulta. Abre un cursor para recuperar resultados de consulta. Recupera una fila de resultados de consulta. Cierra un cursor.

Componentes sintácticos

La mayoría de sentencias SQL tienen la misma estructura.

Todas comienzan por un verbo (select, insert, update, create), a continuación le sigue una o más cláusulas que nos dicen los datos con los que vamos a operar (from, where), algunas de estas son opcionales y otras obligatorias como es el caso del from.



2 - Crear una tabla (create table) DDL

Una base de datos almacena su información en tablas.

Una tabla es una estructura de datos que organiza los datos en columnas y filas; cada columna es un campo (o atributo) y cada fila, un registro. La intersección de una columna con una fila, contiene un dato específico, un solo valor.

Cada registro contiene un dato por cada columna de la tabla.

Cada campo (columna) debe tener un nombre. El nombre del campo hace referencia a la información que almacenará.

Cada campo (columna) también debe definir el tipo de dato que almacenará.

Las tablas forman parte de una base de datos.

Nosotros trabajaremos con la base de datos llamada Temario , que ya he creado en mi equipo

Al crear una tabla debemos resolver qué campos (columnas) tendrá y que tipo de datos almacenarán cada uno de ellos, es decir, su estructura.

La sintaxis básica y general para crear una tabla es la siguiente:

```
create table NOMBRETABLA(  
    NOMBRECAMPO1 TIPODEDATO,  
    ...  
    NOMBRECAMPO1 TIPODEDATO  
)
```

La tabla debe ser definida con un nombre que la identifique y con el cual accederemos a ella.

Creamos una tabla llamada "usuarios" y entre paréntesis definimos los campos y sus tipos:

```
create table usuarios (  
    nombre varchar(30),  
    clave varchar(10)  
)
```

Cada campo con su tipo debe separarse con comas de los siguientes, excepto el último.

Cuando se crea una tabla debemos indicar su nombre y definir al menos un campo con su tipo de dato. En esta tabla "usuarios" definimos 2 campos:

nombre: que contendrá una cadena de caracteres de 30 caracteres de longitud, que almacenará el nombre de usuario y

clave: otra cadena de caracteres de 10 de longitud, que guardará la clave de cada usuario.

Cada usuario ocupará un registro de esta tabla, con su respectivo nombre y clave.

Para nombres de tablas, se puede utilizar cualquier carácter alfabético o numérico, el primero debe ser un carácter alfabético y no puede contener espacios en blanco.

Si intentamos crear una tabla con un nombre ya existente (existe otra tabla con ese nombre), mostrará un mensaje indicando que ya hay un objeto llamado 'usuarios' en la base de datos y la sentencia no se ejecutará. Para evitar problemas cuando estamos haciendo los ejercicios verificaremos primero si ya existe una tabla con dicho nombre y procederemos a borrarla.

Para ver la estructura de una tabla consultaremos una tabla propia del PostgreSQL:

```
SELECT table_name,column_name,udt_name,character_maximum_length  
FROM information_schema.columns WHERE table_name = 'usuarios';
```

Aparece el nombre de la tabla, los nombres de columna y el largo máximo de cada campo.:

table_name	column_name	udt_name	character_maximum_length
usuarios	clave	varchar	10
usuarios	nombre	varchar	30

Para eliminar una tabla usamos "drop table" junto al nombre de la tabla a eliminar:

```
drop table usuarios;
```

Si intentamos eliminar una tabla que no existe, aparece un mensaje de error indicando tal situación y la sentencia no se ejecuta, podemos escribir una variante de drop table verificando si existe la tabla:

```
drop table if exists usuarios;
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
-- Borramos la tabla usuarios si ya existe
```

```
drop table if exists usuarios;
```

```
-- Creamos la tabla usuarios
```

```
create table usuarios (  
    nombre varchar(30),  
    clave varchar(10)
```

```
);
```

-- Mostramos la estructura de la tabla que acabamos de crear

```
select table_name,column_name,udt_name,character_maximum_length  
from information_schema.columns  
where table_name = 'usuarios';
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows the Navicat Premium interface. On the left, the object browser displays a connection to 'PostgreSQL' with a database 'BD1' selected. Under 'BD1', there is a 'public' schema containing a 'Tables' folder with a single entry: 'usuarios'. To the right of the object browser is the main workspace. The top bar has tabs for 'Consulta' (Query) and 'Generador de Consultas' (Query Builder). The bottom bar includes buttons for 'Guardar' (Save), 'Ejecutar seleccionada' (Execute selected), and 'Ejecutar todo' (Execute all). The query editor contains the following SQL code:

```
9  
10 -- Mostramos la estructura de la tabla que acabamos  
de crear  
11 | select table_name,column_name,udt_name,  
character_maximum_length  
12 |     from information_schema.columns  
13 | where table_name = 'usuarios';  
14
```

The results are displayed in a table titled 'Resultado 1' (Result 1). The table has four columns: 'table_name', 'column_name', 'udt_name', and 'character_maximum_length'. It contains two rows:

table_name	column_name	udt_name	character_maximum_length
usuarios	nombre	varchar	30
usuarios	clave	varchar	10

En SQL si queremos disponer un comentario debemos anteceder los caracteres --

-- Esto es un comentario y PostgreSQL lo ignora

3 - Insertar y recuperar registros de una tabla (insert into - select) DML

Un registro es una fila de la tabla que contiene los datos propiamente dichos. Cada registro tiene un dato por cada columna (campo). Nuestra tabla "usuarios" consta de 2 campos, "nombre" y "clave".

Al ingresar los datos de cada registro debe tenerse en cuenta la cantidad y el orden de los campos.

La sintaxis básica y general es la siguiente:

```
insert into NOMBRERECAMPO (NOMBRECAMPO1, ..., NOMBRECAMPOn)  
values (VALORCAMPO1, ..., VALORCAMPOn);
```

Usamos "insert into", luego el nombre de la tabla, detallamos los nombres de los campos entre paréntesis y separados por comas y luego de la cláusula "values" colocamos los valores para cada campo, también entre paréntesis y separados por comas.

Para agregar un registro a la tabla tipeamos:

```
insert into usuarios (nombre, clave) values ('Mariano','payaso');
```

Note que los datos ingresados, como corresponden a cadenas de caracteres se colocan entre comillas simples.

Para ver los registros de una tabla usamos "select":

```
select * from usuarios;
```

El comando "select" recupera los registros de una tabla.

Con el asterisco indicamos que muestre todos los campos de la tabla "usuarios".

Es importante ingresar los valores en el mismo orden en que se nombran los campos:

```
insert into usuarios (clave, nombre) values ('River','Juan');
```

En el ejemplo anterior se nombra primero el campo "clave" y luego el campo "nombre" por eso, los valores también se colocan en ese orden.

Si ingresamos los datos en un orden distinto al orden en que se nombraron los campos, no aparece un mensaje de error y los datos se guardan de modo incorrecto.

En el siguiente ejemplo se colocan los valores en distinto orden en que se nombran los campos, el valor de la clave (la cadena "Boca") se guardará en el campo "nombre" y el valor del nombre (la cadena "Luis") en el campo "clave":

```
insert into usuarios (nombre,clave) values ('Boca','Luis');
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

-- Eliminamos la tabla usuarios si existe

```
drop table if exists usuarios;
```

-- Creamos la tabla

```
create table usuarios(  
    nombre varchar(30),  
    clave varchar(10)  
);
```

-- Agregamos varios registros a la tabla:

```
insert into usuarios (nombre, clave) values ('Mariano','payaso');  
insert into usuarios (nombre, clave) values ('Pablo','jfx344');  
insert into usuarios (nombre, clave) values ('Ana','tru3fal');
```

--Veamos cómo PostgreSQL almacenó los datos:

```
select * from usuarios;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows the Navicat Premium interface. On the left, the Object Browser displays a connection to 'PostgreSQL' with a database 'BD1'. Under 'BD1', there is a 'public' schema containing a 'Tablas' folder with a table named 'usuarios'. The 'Funciones', 'Consultas', 'Informes', and 'Copias de seguridad' folders are also visible. On the right, the main window has a toolbar with icons for Conexión, Nueva Consulta, Tabla, Ver, Vista Materializada, Función, Rol, Otros, Consulta, Informe, and Copia de se. Below the toolbar, a dropdown menu shows 'PostgreSQL' selected. A status bar at the bottom indicates 'BD1 public'. The central area contains a query editor with the following SQL code:

```
'jfx344');  
13 insert into usuarios (nombre, clave) values ('Ana',  
'tru3fal');  
14  
15 --Veamos cómo PostgreSQL almacenó los datos:  
16 select * from usuarios;  
17  
18
```

Below the code, a message tab shows the result of the last command:

nombre	clave
Mariano	payaso
Pablo	jfx344
Ana	tru3fal

4 - Tipos de datos básicos

Ya explicamos que al crear una tabla debemos resolver qué campos (columnas) tendrá y que tipo de datos almacenará cada uno de ellos, es decir, su estructura.

El tipo de dato especifica el tipo de información que puede guardar un campo: caracteres, números, etc.

Estos son algunos tipos de datos básicos de PostgreSQL (posteriormente veremos otros):

varchar: se usa para almacenar cadenas de caracteres. Una cadena es una secuencia de caracteres. Se coloca entre comillas simples; ejemplo: 'Hola', 'Juan Perez'. El tipo "varchar" define una cadena de longitud variable en la cual determinamos el máximo de caracteres entre paréntesis. Puede guardar hasta 10485760 caracteres. Por ejemplo, para almacenar cadenas de hasta 30 caracteres, definimos un campo de tipo varchar(30), es decir, entre paréntesis, junto al nombre del campo colocamos la longitud.

Si asignamos una cadena de caracteres de mayor longitud que la definida, la cadena no se carga, aparece un mensaje indicando tal situación y la sentencia no se ejecuta (ERROR: value too long for type character varying(30)).

Por ejemplo, si definimos un campo de tipo varchar(10) e intentamos asignarle la cadena 'Buenas tardes', aparece un mensaje de error y la sentencia no se ejecuta.

integer: se usa para guardar valores numéricos enteros, de -2000000000 a 2000000000 aprox.

Definimos campos de este tipo cuando queremos representar, por ejemplo, cantidades.

float: se usa para almacenar valores numéricos con decimales. Se utiliza como separador el punto (.). Definimos campos de este tipo para precios, por ejemplo.

Antes de crear una tabla debemos pensar en sus campos y optar por el tipo de dato adecuado para cada uno de ellos.

Por ejemplo, si en un campo almacenaremos números enteros, el tipo "float" sería una mala elección; si vamos a guardar precios, el tipo "float" es más adecuado, no así "integer" que no tiene decimales. Otro ejemplo, si en un campo vamos a guardar un número telefónico o un número de documento, usamos "varchar", no "integer" porque si bien son dígitos, con ellos no realizamos operaciones matemáticas.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
    titulo varchar(20),
    autor varchar(15),
    editorial varchar(10),
```

```

precio float,
cantidad integer
);

```

```

select table_name,column_name,udt_name,character_maximum_length
from information_schema.columns
where table_name = 'libros';

```

```

insert into libros (titulo,autor,editorial,precio,cantidad)
values ('El aleph','Borges','Emece',25.50,100);

insert into libros (titulo,autor,editorial,precio,cantidad)
values ('Matematica estas ahi','Paenza','Siglo XXI',18.8,200);

```

```
select * from libros;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows the Navicat Premium interface. On the left, the object browser displays a connection to 'CONEXION AL POSTGRES DE BD1' with a 'public' schema selected. In the center, a query editor window titled 'Sin título - Consulta' contains the following SQL code:

```

cantidad)
18 values ('Matematica estas ahi','Paenza','Siglo XXI',
18.8,200);
19
20 | select * from libros;
21
22
23

```

At the bottom, the 'Resultado 1' tab shows the execution result:

titulo	autor	editorial	precio	cantidad
El aleph	Borges	Emece	25,5	100
Matematica estas ahi	Paenza	Siglo XXI	18,8	200

Si tenemos varios comandos "select" en el lote de comandos en la pestaña "Resultado 1" se muestra el resultado del último. Si queremos ver el resultado de otro "select" debemos seleccionarlo con el mouse y presionar nuevamente el ícono de **ejecutar seleccionados**

para que se ejecute solo ese comando:

* Sin título - Consulta - Navicat Premium

Archivo Editar Ver Consulta Formato Favoritos Herramientas Ventanas Ayuda

Conexión Nueva Consulta Tabla Ver Vista Materializada Función Rol Otros Consulta Informe Copia de seguridad

MySQL CONEXION AL POSTGRES DE PostgreSQL BD1 public Tablas usuarios Vistas Vistas materializadas Funciones Consultas Informes Copias de seguridad biblioteca BIBLIOTECAX ejercicios FACTURACION HOSPITAL importaciones

Objetos * Sin título - Consulta Guardar Generador de Consultas Embellecer SQL Fragmento de código Texto Ejecutar seleccionados

PostgreSQL BD1 public

```
8   cantidad integer
9 );
10
11 | select table_name,column_name,udt_name,
12 |       character_maximum_length
13 |   from information_schema.columns
14 | where table_name = 'libros';
```

Mensaje Resultado 1

titulo	autor	editorial	precio	cantidad
El aleph	Borges	Emece	25,5	100
Matematica estas ahí	Paenza	Siglo XXI	18,8	200

5 - Recuperar algunos campos (select) DML

Hemos aprendido cómo ver todos los registros de una tabla, empleando la instrucción "select".

La sintaxis básica y general es la siguiente:

```
select * from NOMBRETABLA;
```

El asterisco (*) indica que se seleccionan todos los campos de la tabla.

Podemos especificar el nombre de los campos que queremos ver separándolos por comas:

```
select titulo,autor from libros;
```

La lista de campos luego del "select" selecciona los datos correspondientes a los campos nombrados. En el ejemplo anterior seleccionamos los campos "titulo" y "autor" de la tabla "libros", mostrando todos los registros. Los datos aparecen ordenados según la lista de selección, en dicha lista los nombres de los campos se separan con comas.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
    titulo varchar(40),
```

```
    autor varchar(30),
```

```
    editorial varchar(15),
```

```
    precio float,
```

```
    cantidad integer
```

```
);
```

```
select table_name,column_name,udt_name,character_maximum_length
```

```
from information_schema.columns
```

```
where table_name = 'libros';
```

```
insert into libros (titulo,autor,editorial,precio,cantidad)
```

```
values ('El aleph','Borges','Emece',25.50,100);
```

```
insert into libros (titulo,autor,editorial,precio,cantidad)
```

```
values ('Alicia en el pais de las maravillas','Lewis Carroll','Atlantida',10,200);
```

```
insert into libros (titulo,autor,editorial,precio,cantidad)
```

```
values ('Matematica estas ahi','Paenza','Siglo XXI',18.8,200);
```

```
select * from libros;
```

```
select titulo,autor,editorial from libros;
```

```
select titulo,precio from libros;
```

```
select editorial,cantidad from libros;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
* Sin título - Consulta - Navicat Premium

Archivo Editar Ver Consulta Formato Favoritos Herramientas Ventanas Ayuda
Conexión Nueva Consulta Tabla Ver Vista Materializada Función Rol Otros Consulta Informe Copia de s
MySQL CONEXION AL POSTGRES DE PostgreSQL BD1 public Tablas usuarios Vistas Vistas materializadas Funciones Consultas Informes Copias de seguridad biblioteca BIBLIOTECAX ejercicios FACTURACION HOSPITAL importaciones miracle odoo
Objetos * Sin título - Consulta
Guarda Generador de Consultas Embellecer SQL Fragmento de código Texto
PostgreSQL BD1 public Ejecutar seleccionados
24 select titulo,autor,editorial from libros;
25
26 select titulo,precio from libros;
27
28 select editorial,cantidad from libros;
29
30
31
Mensaje Resultado 1
editorial cantidad
Emece 100
Atlantida 200
Siglo XXI 200
```

Recordemos que en la pestaña "Data Output" se muestra el resultado del último comando 'select' ejecutado. Si queremos ver el resultado de otro 'select' debemos seleccionarlo y luego volver a ejecutar:

* Sin título - Consulta - Navicat Premium

Archivo Editar Ver Consulta Formato Favoritos Herramientas Ventanas Ayuda

Conexión Nueva Consulta Tabla Ver Vista Materializada Función Rol Otros Consulta Informe Copia de s

MySQL CONEXION AL POSTGRES DE PostgreSQL BD1 public Tablas usuarios Vistas Vistas materializadas Funciones Consultas Informes Copias de seguridad biblioteca BIBLIOTECAX ejercicios FACTURACION HOSPITAL importaciones miracle

Objetos * Sin título - Consulta Guardar Generador de Consultas Embellecer SQL Fragmento de código Texto Postgre SQL BD1 public Ejecutar seleccionados

```
24 select titulo,autor,editorial from libros;
25
26 select titulo,precio from libros;
27
28 select editorial,cantidad from libros;
29
30
31
```

Mensaje Resultado 1

titulo	precio
El aleph	25,5
Alicia en el país de las maravillas	10
Matemática estas ahí	18,8

6 - Recuperar algunos registros (where)

Hemos aprendido a seleccionar algunos campos de una tabla.

También es posible recuperar algunos registros.

Existe una cláusula, "where" con la cual podemos especificar condiciones para una consulta "select". Es decir, podemos recuperar algunos registros, sólo los que cumplan con ciertas condiciones indicadas con la cláusula "where". Por ejemplo, queremos ver el usuario cuyo nombre es "Marcelo", para ello utilizamos "where" y luego de ella, la condición:

```
select nombre, clave  
from usuarios  
where nombre='Marcelo';
```

La sintaxis básica y general es la siguiente:

```
select NOMBRECAMPO1, ..., NOMBRECAMPOn  
from NOMBRETABLA  
where CONDICION;
```

Para las condiciones se utilizan operadores relacionales (tema que trataremos más adelante en detalle). El signo igual(=) es un operador relacional.

Para la siguiente selección de registros especificamos una condición que solicita los usuarios cuya clave es igual a "River":

```
select nombre,clave  
from usuarios  
where clave='River';
```

Si ningún registro cumple la condición establecida con el "where", no aparecerá ningún registro.

Entonces, con "where" establecemos condiciones para recuperar algunos registros.

Para recuperar algunos campos de algunos registros combinamos en la consulta la lista de campos y la cláusula "where":

```
select nombre  
from usuarios  
where clave='River';
```

En la consulta anterior solicitamos el nombre de todos los usuarios cuya clave sea igual a "River".

Ingresemos el siguiente lote de comandos SQL en navicat:

```
drop table if exists usuarios;
```

```
create table usuarios (
    nombre varchar(30),
    clave varchar(10)
);
```

-- Vemos la estructura de la tabla:

```
select table_name,column_name,udt_name,character_maximum_length
from information_schema.columns
where table_name = 'usuarios';
```

-- Ingresamos algunos registros:

```
insert into usuarios (nombre, clave)
values ('Marcelo','Boca');

insert into usuarios (nombre, clave)
values ('JuanPerez','Juancito');

insert into usuarios (nombre, clave)
values ('Susana','River');

insert into usuarios (nombre, clave)
values ('Luis','River');
```

-- Realizamos una consulta especificando una condición, queremos

-- ver el usuario cuyo nombre es "Leonardo":

```
select * from usuarios
where nombre='Leonardo';
```

-- Realizamos un "select" de los nombres de los usuarios cuya clave es "Santi":

```
select nombre from usuarios
where clave='Santi';
```

-- Queremos ver el nombre de los usuarios cuya clave es "River":

```
select nombre from usuarios
```

```
where clave='River';
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows the Navicat Premium interface for PostgreSQL. The left sidebar lists databases: MySQL, CONEXION AL POSTGRES DE, Postgre SQL, BD1 (selected), public, Tablas (selected), usuarios, Vistas, Vistas materializadas, Funciones, Consultas, Informes, and Copias de seguridad. Other databases listed are biblioteca, BIBLIOTECAX, ejercicios, FACTURACION, HOSPITAL, importaciones, and miracle. The main window has a toolbar with Table, View, Materialized View, Function, Role, Others, Query, Report, and Copy buttons. A menu bar includes Archivo, Editar, Ver, Consulta, Formato, Favoritos, Herramientas, Ventanas, and Ayuda. A top bar shows the connection as * Sin título - Consulta - Navicat Premium. The query editor contains the following code:

```
28 select nombre from usuarios
29 where clave='Santi';
30
31 -- Queremos ver el nombre de los usuarios cuya clave
32 es "River":
33 select nombre from usuarios
34 where clave='River';
```

The results pane shows a table with one row:

nombre
Susana
Luis

7 - Operadores relacionales

Los operadores relacionales (o de comparación) nos permiten comparar dos expresiones, que pueden ser variables, valores de campos, etc.

Hemos aprendido a especificar condiciones de igualdad para seleccionar registros de una tabla; por ejemplo:

```
select * from libros  
where autor='Borges';
```

Utilizamos el operador relacional de igualdad.

Los operadores relacionales vinculan un campo con un valor para que PostgreSQL compare cada registro (el campo especificado) con el valor dado.

Los operadores relacionales son los siguientes:

= igual	like
<> distinto	Not Like
> mayor	is Null / Is not Null
< menor	is between / is not between
>= mayor o igual	is in list / is not in list
<= menor o igual	

Podemos seleccionar los registros cuyo autor sea diferente de "Borges", para ello usamos la condición:

```
select * from libros  
where autor<>'Borges';
```

Podemos comparar valores numéricos. Por ejemplo, queremos mostrar los títulos y precios de los libros cuyo precio sea mayor a 20 euros:

```
select titulo, precio  
from libros  
where precio>20;
```

Queremos seleccionar los libros cuyo precio sea menor o igual a 30:

```
select * from libros
```

```
where precio<=30;
```

Los operadores relacionales comparan valores del mismo tipo. Se emplean para comprobar si un campo cumple con una condición.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
    titulo varchar(30),
```

```
    autor varchar(30),
```

```
    editorial varchar(15),
```

```
    precio float
```

```
);
```

```
insert into libros (titulo,autor,editorial,precio)
```

```
values ('El aleph','Borges','Emece',24.50);
```

```
insert into libros (titulo,autor,editorial,precio)
```

```
values ('Martin Fierro','Jose Hernandez','Emece',16.00);
```

```
insert into libros (titulo,autor,editorial,precio)
```

```
values ('Aprenda PHP','Mario Molina','Emece',35.40);
```

```
insert into libros (titulo,autor,editorial,precio)
```

```
values ('Cervantes y el quijote','Borges','Paidos',50.90);
```

```
-- Seleccionamos los registros cuyo autor sea diferente de 'Borges':
```

```
select * from libros
```

```
where autor<>'Borges';
```

```
-- Seleccionamos los registros cuyo precio supere los 20 euros, sólo el título y precio:
```

```
select titulo,precio
```

```
from libros  
where precio>20;
```

-- Recuperamos aquellos libros cuyo precio es menor o igual a 30:

```
select * from libros  
where precio<=30;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL 11*  
values ('El aleph','Borges','Emece',24.50);  
insert into libros (titulo,autor,editorial,precio)  
values ('Martin Fierro','Jose Hernandez','Emece',16.00);  
insert into libros (titulo,autor,editorial,precio)  
values ('Aprenda PHP','Mario Molina','Emece',35.40);  
insert into libros (titulo,autor,editorial,precio)  
values ('Cervantes y el quijote','Borges','Paidos',50.90);  
  
-- Seleccionamos los registros cuyo autor sea diferente de 'Borges':  
select * from libros  
where autor<>'Borges';  
  
-- Seleccionamos los registros cuyo precio supere los 20 pesos, sólo el titulo y precio:  
select titulo,precio  
from libros  
where precio>20;  
  
-- Recuperamos aquellos libros cuyo precio es menor o igual a 30:  
select * from libros  
where precio<=30;
```

	titulo	autor	editorial	precio
	character varying (30)	character varying (30)	character varying (15)	double precision
1	El aleph	Borges	Emece	24.5
2	Martin Fierro	Jose Hernandez	Emece	16

8 - Borrar registros (delete) DML

Para eliminar los registros de una tabla usamos el comando "delete":

```
delete from usuarios;
```

Si no queremos eliminar todos los registros, sino solamente algunos, debemos indicar cuál o cuáles, para ello utilizamos el comando "delete" junto con la cláusula "where" con la cual establecemos la condición que deben cumplir los registros a borrar.

Por ejemplo, queremos eliminar aquel registro cuyo nombre de usuario es "Marcelo":

```
delete from usuarios  
where nombre='Marcelo';
```

Si solicitamos el borrado de un registro que no existe, es decir, ningún registro cumple con la condición especificada, ningún registro será eliminado.

Tenga en cuenta que si no colocamos una condición, se eliminan todos los registros de la tabla nombrada.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists usuarios;
```

```
create table usuarios(  
    nombre varchar(30),  
    clave varchar(10)  
);
```

```
insert into usuarios (nombre,clave)  
values ('Marcelo','River');  
insert into usuarios (nombre,clave)  
values ('Susana','chapita');  
insert into usuarios (nombre,clave)  
values ('CarlosFuentes','Boca');  
insert into usuarios (nombre,clave)
```

```
values ('FedericoLopez','Boca');
```

```
select * from usuarios;
```

-- Vamos a eliminar el registro cuyo nombre de usuario es "Marcelo":

```
delete from usuarios
```

```
where nombre='Marcelo';
```

-- Veamos el contenido de la tabla:

```
select * from usuarios;
```

-- Intentamos eliminarlo nuevamente:

```
delete from usuarios
```

```
where nombre='Marcelo';
```

```
select * from usuarios;
```

-- Eliminamos todos los registros cuya clave es 'Boca':

```
delete from usuarios
```

```
where clave='Boca';
```

-- Veamos el contenido de la tabla:

```
select * from usuarios;
```

-- Eliminemos todos los registros:

```
delete from usuarios;
```

-- Veamos el contenido de la tabla:

```
select * from usuarios;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL query editor interface. At the top, there are tabs for Dashboard, Properties, SQL, Statistics, Dependencies, Dependents, and Query. Below the tabs, there is a toolbar with icons for file operations, search, and refresh. A dropdown menu shows 'No limit' selected. The main area contains a SQL script with numbered comments and statements. Lines 32 through 44 are shown, with line 42 highlighted by a red box. The output section below shows a table with two columns: 'nombre' and 'clave'. The first row has 'dave' in the 'nombre' column and 'character varying (10)' in the 'clave' column, also highlighted by a red box.

```
bd1 on postgres@PostgreSQL 11
32 -- Eliminamos todos los registros cuya clave es 'Boca':
33 delete from usuarios
34 where clave='Boca';
35
36 -- Veamos el contenido de la tabla:
37 select * from usuarios;
38
39 -- Eliminemos todos los registros:
40 delete from usuarios;
41
42 -- Veamos el contenido de la tabla:
43 select * from usuarios;
```

nombre	clave
dave	character varying (10)

9 - Actualizar registros (update) DML

Decimos que actualizamos un registro cuando modificamos alguno de sus valores.

Para modificar uno o varios datos de uno o varios registros utilizamos "update" (actualizar).

Por ejemplo, en nuestra tabla "usuarios", queremos cambiar los valores de todas las claves, por "RealMadrid":

```
update usuarios set clave='RealMadrid';
```

Utilizamos "update" junto al nombre de la tabla y "set" junto con el campo a modificar y su nuevo valor.

El cambio afectará a todos los registros.

Podemos modificar algunos registros, para ello debemos establecer condiciones de selección con "where".

Por ejemplo, queremos cambiar el valor correspondiente a la clave de nuestro usuario llamado "Federicolopez", queremos como nueva clave "Boca", necesitamos una condición "where" que afecte solamente a este registro:

```
update usuarios set clave='Boca'  
where nombre='Federicolopez';
```

Si PostgreSQL no encuentra registros que cumplan con la condición del "where", no se modifica ninguno.

Las condiciones no son obligatorias, pero si omitimos la cláusula "where", la actualización afectará a todos los registros.

También podemos actualizar varios campos en una sola instrucción:

```
update usuarios set nombre='Marceloduarte', clave='Marce'  
where nombre='Marcelo';
```

Para ello colocamos "update", el nombre de la tabla, "set" junto al nombre del campo y el nuevo valor y separado por coma, el otro nombre del campo con su nuevo valor.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists usuarios;
```

```
create table usuarios(
```

```
    nombre varchar(20),
```

```
    clave varchar(10)
```

```
);
```

```
insert into usuarios (nombre,clave)  
values ('Marcelo','River');  
  
insert into usuarios (nombre,clave)  
values ('Susana','chapita');  
  
insert into usuarios (nombre,clave)  
values ('Carlosfuentes','Boca');  
  
insert into usuarios (nombre,clave)  
values ('Federicolopez','Boca');
```

-- Cambiaremos los valores de todas las claves, por la cadena "RealMadrid":

```
update usuarios set clave='RealMadrid';
```

-- El cambio afectó a todos los registros, veámoslo:

```
select * from usuarios;
```

-- Necesitamos cambiar el valor de la clave del usuario llamado "Federicolopez" por "Boca":

```
update usuarios set clave='Boca'  
where nombre='Federicolopez';
```

-- Verifiquemos que la actualización se realizó:

```
select * from usuarios;
```

-- Vimos que si PostgreSQL no encuentra registros que cumplan con la condición no se

-- modifican registros:

```
update usuarios set clave='payaso'  
where nombre='JuanaJuarez';
```

```
select * from usuarios;
```

-- Para actualizar varios campos en una sola instrucción empleamos:

```
update usuarios set nombre='Marceloduarte', clave='Marce'  
where nombre='Marcelo';
```

-- Si vemos la tabla:

```
select * from usuarios;
```

Seleccionemos con el mouse todas las instrucciones hasta el primer comando 'select' y luego ejecutemos dicho bloque. Debe aparecer una salida similar a:

```
bd1 on postgres@PostgreSQL 11  
1 drop table if exists usuarios;  
2  
3 create table usuarios(  
4     nombre varchar(20),  
5     clave varchar(10)  
6 );  
7  
8 insert into usuarios (nombre,clave)  
9 values ('Marcelo','River');  
10 insert into usuarios (nombre,clave)  
11 values ('Susana','chapita');  
12 insert into usuarios (nombre,clave)  
13 values ('Carlosfuentes','Boca');  
14 insert into usuarios (nombre,clave)  
15 values ('Federicolopez','Boca');  
16  
17 -- Cambiaremos los valores de todas las claves, por la cadena "RealMadrid":  
18 update usuarios set clave='RealMadrid';  
19  
20 -- El cambio afectó a todos los registros, veámoslo:  
21 select * from usuarios;  
22  
23 -- Necesitamos cambiar el valor de la clave del usuario llamado "Federicolopez" por "Boca":  
24 update usuarios set clave='Boca'  
25 where nombre='Federicolopez';  
26
```

	nombre	clave
1	Marcelo	RealMadrid
2	Susana	RealMadrid
3	Carlosfuentes	RealMadrid
4	Federicolopez	RealMadrid

Ejecutemos luego los otros update y veamos cada uno de los resultados.

10 - Comentarios

Para aclarar algunas instrucciones, en ocasiones, necesitamos agregar comentarios.

Es posible ingresar comentarios en la línea de comandos, es decir, un texto que no se ejecuta; para ello se emplean dos guiones (--) al comienzo de la línea:

```
select * from libros -- mostramos los registros de libros;
```

en la línea anterior, todo lo que está luego de los guiones (hacia la derecha) no se ejecuta.

Para agregar varias líneas de comentarios, se coloca una barra seguida de un asterisco /*) al comienzo del bloque de comentario y al finalizarlo, un asterisco seguido de una barra (*/).

```
select titulo, autor
```

```
/*mostramos títulos y
```

```
nombres de los autores*/
```

```
from libros;
```

todo lo que está entre los símbolos /*" y "*/ no se ejecuta.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
-- Eliminamos la tabla usuarios si existe
```

```
drop table if exists usuarios;
```

```
-- Creamos la tabla
```

```
create table usuarios(
```

```
nombre varchar(30),
```

```
clave varchar(10)
```

```
);
```

```
/*
```

Otra estructura de la tabla usuarios con el campo mail

```
create table usuarios(
```

```
nombre varchar(30),
```

```
clave varchar(10),
```

```

    mail varchar(70)
);

*/

```

-- Agregamos varios registros a la tabla:

```

insert into usuarios (nombre, clave) values ('Mariano','payaso');

insert into usuarios (nombre, clave) values ('Pablo','jfx344');

insert into usuarios (nombre, clave) values ('Ana','tru3fal');

```

--Veamos cómo PostgreSQL almacenó los datos:

```
select * from usuarios; -- Muestra todos los campos
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL query editor interface with the following details:

- Toolbar:** Includes tabs for Dashboard, Properties, SQL, Statistics, Dependencies, Dependents, and Query - bc.
- Query Editor:** Displays the following SQL code:


```

1 -- Eliminamos la tabla usuarios si existe
2 drop table if exists usuarios;

3 -- Creamos la tabla
4 create table usuarios(
5   nombre varchar(30),
6   clave varchar(10)
7 );
8
9
10 /*
11 otra estructura de la tabla usuarios con el campo mail
12 create table usuarios(
13   nombre varchar(30),
14   clave varchar(10),
15   mail varchar(70)
16 );
17 */
18
19 -- Agregamos varios registros a la tabla:
20 insert into usuarios (nombre, clave) values ('Mariano','payaso');
21 insert into usuarios (nombre, clave) values ('Pablo','jfx344');
22 insert into usuarios (nombre, clave) values ('Ana','tru3fal');
23
24 --Veamos cómo PostgreSQL almacenó los datos:
25 select * from usuarios; -- Muestra todos los campos
      
```
- Result Table:** Shows the data from the 'usuarios' table.

	nombre	clave
1	Mariano	payaso
2	Pablo	jfx344
3	Ana	tru3fal

11 - Valores null (is null)

"null" significa "dato desconocido" o "valor inexistente". No es lo mismo que un valor "0", una cadena vacía o una cadena literal "null".

A veces, puede desconocerse o no existir el dato correspondiente a algún campo de un registro. En estos casos decimos que el campo puede contener valores nulos.

Por ejemplo, en nuestra tabla de libros, podemos tener valores nulos en el campo "precio" porque es posible que para algunos libros no le hayamos establecido el precio para la venta.

En contraposición, tenemos campos que no pueden estar vacíos jamás.

Veamos un ejemplo. Tenemos nuestra tabla "libros". El campo "titulo" no debería estar vacío nunca, igualmente el campo "autor". Para ello, al crear la tabla, debemos especificar que dichos campos no admitan valores nulos:

```
create table libros(  
    titulo varchar(30) not null,  
    autor varchar(20) not null,  
    editorial varchar(15) null,  
    precio float  
);
```

Para especificar que un campo no admite valores nulos, debemos colocar "not null" luego de la definición del campo.

En el ejemplo anterior, los campos "editorial" y "precio" si admiten valores nulos.

Cuando colocamos "null" estamos diciendo que admite valores nulos (caso del campo "editorial"); por defecto, es decir, si no lo aclaramos, los campos permiten valores nulos (caso del campo "precio").

Si ingresamos los datos de un libro, para el cual aún no hemos definido el precio podemos colocar "null" para mostrar que no tiene precio:

```
insert into libros (titulo,autor,editorial,precio)  
values('El aleph','Borges','Emece',null);
```

Note que el valor "null" no es una cadena de caracteres, no se coloca entre comillas.

Entonces, si un campo acepta valores nulos, podemos ingresar "null" cuando no conocemos el valor.

También podemos colocar "null" en el campo "editorial" si desconocemos el nombre de la editorial a la cual pertenece el libro que vamos a ingresar:

```
insert into libros (titulo,autor,editorial,precio)
```

```
values('Alicia en el pais','Lewis Carroll',null,25);
```

Si intentamos ingresar el valor "null" en campos que no admiten valores nulos (como "titulo" o "autor"), PostgreSQL no lo permite, muestra un mensaje y la inserción no se realiza; por ejemplo:

```
insert into libros (titulo,autor,editorial,precio)
```

```
values(null,'Borges','Siglo XXI',25);
```

Para ver cuáles campos admiten valores nulos y cuáles no, podemos consultar el catálogo. Nos muestra mucha información, en la columna "is_nullable" vemos que muestra "NO" en los campos que no permiten valores nulos y "YES" en los campos que si los permiten.

```
select table_name,column_name,udt_name,character_maximum_length,is_nullable
```

```
from information_schema.columns
```

```
where table_name = 'libros';
```

Para recuperar los registros que contengan el valor "null" en algún campo, no podemos utilizar los operadores relacionales vistos anteriormente: = (igual) y <> (distinto); debemos utilizar los operadores "is null" (es igual a null) y "is not null" (no es null):

```
select * from libros
```

```
where precio is null;
```

La sentencia anterior tendrá una salida diferente a la siguiente:

```
select * from libros
```

```
where precio=0;
```

Con la primera sentencia veremos los libros cuyo precio es igual a "null" (desconocido); con la segunda, los libros cuyo precio es 0.

Igualmente para campos de tipo cadena, las siguientes sentencias "select" no retornan los mismos registros:

```
select * from libros where editorial is null;
```

```
select * from libros where editorial='';
```

Con la primera sentencia veremos los libros cuya editorial es igual a "null", con la segunda, los libros cuya editorial guarda una cadena vacía.

Entonces, para que un campo no permita valores nulos debemos especificarlo luego de definir el campo, agregando "not null". Por defecto, los campos permiten valores nulos, pero podemos especificarlo igualmente agregando "null".

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(  
    titulo varchar(30) not null,  
    autor varchar(30) not null,  
    editorial varchar(15) null,  
    precio float  
);  
  
insert into libros (titulo,autor,editorial,precio)  
values('El aleph','Borges','Emece',null);  
  
insert into libros (titulo,autor,editorial,precio)  
values('Alicia en el pais','Lewis Carroll',null,0);
```

--Para ver cuáles campos admiten valores nulos y cuáles no, empleamos:

```
select table_name,column_name,udt_name,character_maximum_length,is_nullable  
from information_schema.columns  
where table_name = 'libros';
```

-- Dijimos que el valor "null" no es lo mismo que una cadena vacía.

-- Vamos a ingresar un registro con cadena vacía para el campo "editorial":

```
insert into libros (titulo,autor,editorial,precio)  
values('Uno','Richard Bach','','18.50');
```

-- Ingresamos otro registro, ahora cargamos una cadena vacía en el campo "titulo":

```
insert into libros (titulo,autor,editorial,precio)  
values('','Richard Bach','Planeta',22);
```

-- Veamos todos los registros ingresados:

```
select * from libros;
```

-- Recuperaremos los registros que contengan el valor "null" en el campo "precio":

```
select * from libros
```

```
where precio is null;
```

```
select * from libros
```

```
where precio=0;
```

-- Recuperemos los libros cuyo nombre de editorial es "null":

```
select * from libros
```

```
where editorial is null;
```

-- Ahora veamos los libros cuya editorial almacena una cadena vacía:

```
select * from libros
```

```
where editorial='';
```

-- Para recuperar los libros cuyo precio no sea nulo tipeamos:

```
select * from libros
```

```
where precio is not null;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
39 -- Recuperemos los libros cuyo nombre de editorial es "null":  
40 select * from libros  
41 where editorial is null;  
42  
43 -- Ahora veamos los libros cuya editorial almacena una cadena vacía:  
44 select * from libros  
45 where editorial='';  
46  
47 -- Para recuperar los libros cuyo precio no sea nulo tipeamos:  
48 select * from libros  
49 where precio is not null;
```

	título	autor	editorial	precio
1	Alicia en el país.	Lewis Carroll	[null]	0
2	Uno	Richard Bach		18.5
3		Richard Bach	Planeta	22

12 - Clave primaria

Una clave primaria es un campo (o varios) que identifica un solo registro (fila) en una tabla.

Para un valor del campo clave existe solamente un registro.

Veamos un ejemplo, si tenemos una tabla con datos de personas, el número de documento puede establecerse como clave primaria, es un valor que no se repite; puede haber personas con igual apellido y nombre, incluso el mismo domicilio (padre e hijo por ejemplo), pero su documento será siempre distinto.

Si tenemos la tabla "usuarios", el nombre de cada usuario puede establecerse como clave primaria, es un valor que no se repite; puede haber usuarios con igual clave, pero su nombre de usuario será siempre diferente.

Podemos establecer que un campo sea clave primaria al momento de crear la tabla o luego que ha sido creada. Vamos a aprender a establecerla al crear la tabla. Hay 2 maneras de hacerlo, por ahora veremos la sintaxis más sencilla.

Tenemos nuestra tabla "usuarios" definida con 2 campos ("nombre" y "clave").

La sintaxis básica y general es la siguiente:

```
create table NOMBRERATABLA(
```

```
    CAMPO TIPO,
```

```
    ...
```

```
    primary key (NOMBRECAMPO)
```

```
);
```

En el siguiente ejemplo definimos una clave primaria, para nuestra tabla "usuarios" para asegurarnos que cada usuario tendrá un nombre diferente y único:

```
create table usuarios(
```

```
    nombre varchar(20),
```

```
    clave varchar(10),
```

```
    primary key(nombre)
```

```
);
```

Lo que hacemos agregar luego de la definición de cada campo, "primary key" y entre paréntesis, el nombre del campo que será clave primaria.

Una tabla sólo puede tener una clave primaria. Cualquier campo (de cualquier tipo) puede ser clave primaria, debe cumplir como requisito, que sus valores no se repitan ni sean nulos. Por ello, al definir un campo como clave primaria, automáticamente PostgreSQL lo convierte a "not null".

Luego de haber establecido un campo como clave primaria, al ingresar los registros, PostgreSQL controla que los valores para el campo establecido como clave primaria no estén repetidos en la

tabla; si estuviesen repetidos, muestra un mensaje y la inserción no se realiza. Es decir, si en nuestra tabla "usuarios" ya existe un usuario con nombre "juanperez" e intentamos ingresar un nuevo usuario con nombre "juanperez", aparece un mensaje y la instrucción "insert" no se ejecuta.

Igualmente, si realizamos una actualización, PostgreSQL controla que los valores para el campo establecido como clave primaria no estén repetidos en la tabla, si lo estuviese, aparece un mensaje indicando que se viola la clave primaria y la actualización no se realiza.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists usuarios;

-- Creamos la tabla definiendo el campo "nombre" como clave primaria:

create table usuarios(
    nombre varchar(20),
    clave varchar(10),
    primary key(nombre)
);

-- Al campo "nombre" no lo definimos "not null", pero al establecerse como clave primaria,
-- PostgreSQL lo convierte en "not null", veamos que en la columna "is_nullable" aparece "NO":

select table_name,column_name,udt_name,character_maximum_length,is_nullable
from information_schema.columns
where table_name = 'usuarios';

-- Ingresamos algunos registros:

insert into usuarios (nombre, clave)
values ('juanperez','Boca');

insert into usuarios (nombre, clave)
values ('raulgarcia','River');

select * from usuarios;
```

/* Recordemos que cuando un campo es clave primaria, sus valores no se repiten.

Intentamos ingresar un valor de clave primaria existente:

```
insert into usuarios (nombre, clave)  
values ('juanperez','payaso');
```

aparece un mensaje de error y la sentencia no se ejecuta.

Cuando un campo es clave primaria, sus valores no pueden ser nulos.

Intentamos ingresar el valor "null" en el campo clave primaria:

```
insert into usuarios (nombre, clave)  
values (null,'payaso');
```

aparece un mensaje de error y la sentencia no se ejecuta.

Si realizamos alguna actualización, PostgreSQL controla que los valores para el campo establecido como clave primaria no estén repetidos en la tabla.

Intentemos actualizar el nombre de un usuario colocando un nombre existente:

```
update usuarios set nombre='juanperez'  
where nombre='raulgarcia';  
*/
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows the pgAdmin 4 interface with a query editor window. The title bar says 'bd1 on postgres@PostgreSQL 11'. The editor contains the following SQL code:

```
1 drop table if exists usuarios;  
2  
3 -- Creamos la tabla definiendo el campo "nombre" como clave primaria:  
4 create table usuarios(  
5     nombre varchar(20),  
6     clave varchar(10),  
7     primary key(nombre)  
8 );  
9  
10 -- Al campo "nombre" no lo definimos "not null", pero al establecerse como clave primaria,  
11 -- PostgreSQL lo convierte en "not null", veamos que en la columna "is_nullable" aparece "NO":  
12 select table_name,column_name,udt_name,character_maximum_length,is_nullable  
from information_schema.columns  
where table_name = 'usuarios';  
13  
14 -- Ingresamos algunos registros:  
15 insert into usuarios (nombre, clave)  
16     values ('juanperez','Boca');  
17 insert into usuarios (nombre, clave)  
18     values ('raulgarcia','River');  
19  
20 select * from usuarios;
```

Below the code, there is a toolbar with tabs: Data Output, Explain, Messages, Notifications, and Query History. The Data Output tab is selected. A results grid shows the data from the 'usuarios' table:

nombre	clave
juanperez	Boca
raulgarcia	River

13 - Campo entero serial (autoincremento)

Los valores de un campo serial, se inician en 1 y se incrementan en 1 automáticamente.

Se utiliza generalmente en campos correspondientes a códigos de identificación para generar valores únicos para cada nuevo registro que se inserta.

Normalmente se define luego este campo como clave primaria.

Para establecer que un campo autoincremente sus valores automáticamente, éste debe ser de tipo serial:

```
create table libros(
```

```
    codigo serial,
```

```
    titulo varchar(20),
```

```
    autor varchar(30),
```

```
    editorial varchar(15),
```

```
    primary key (codigo)
```

```
);
```

Para definir un campo autoincrementable lo hacemos con la palabra clave serial.

Hasta ahora, al ingresar registros, colocamos el nombre de todos los campos antes de los valores; es posible ingresar valores para algunos de los campos de la tabla, pero recuerde que al ingresar los valores debemos tener en cuenta los campos que detallamos y el orden en que lo hacemos.

Cuando un campo es de tipo serial no es necesario ingresar valor para él, porque se inserta automáticamente tomando el último valor como referencia, o 1 si es el primero.

Para ingresar registros omitimos el campo definido como serial, por ejemplo:

```
insert into libros (titulo,autor,editorial)
```

```
values('El aleph','Borges','Planeta');
```

Este primer registro ingresado guardará el valor 1 en el campo correspondiente al código.

Si continuamos ingresando registros, el código (dato que no ingresamos) se cargará automáticamente siguiendo la secuencia de autoincremento.

Más adelante cuando veamos el concepto de secuencias veremos que los campos serial son en realidad un campo int que tienen asociado una "secuencia".

Un campo serial podemos indicar el valor en el insert, pero en la siguiente inserción que hagamos la secuencia continúa en el último valor generado automáticamente. Puede tener sentido utilizar

esta característica para reutilizar un código de un campo borrado, por ejemplo si en la tabla libros borramos el libro con el código 1, luego podemos insertar otro libro con dicho código:

```
delete from libros where codigo=1;
```

```
insert into libros (codigo,titulo,autor,editorial)
```

```
values(1,'Aprender Python', 'Rodriguez Luis', 'Paidos');
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
    codigo serial,
```

```
    titulo varchar(30),
```

```
    autor varchar(30),
```

```
    editorial varchar(15),
```

```
    primary key (codigo)
```

```
);
```

-- Al visualizar la estructura de la tabla:

```
select table_name,column_name,udt_name,character_maximum_length,is_nullable  
from information_schema.columns  
where table_name = 'libros';
```

-- Ingresamos algunos registros:

```
insert into libros (titulo,autor,editorial)
```

```
values('El aleph','Borges','Planeta');
```

-- Note que al detallar los campos para los cuales ingresaremos valores hemos

-- omitido "codigo"; cuando un campo es serial no es necesario ingresar valor

-- para él, porque se genera automáticamente. Recuerde que si es obligatorio

-- ingresar los datos de todos los campos que se detallan y en el mismo orden.

-- Si mostramos los registros:

```
select * from libros;
```

-- vemos que este primer registro ingresado guardó el valor 1 en el campo

-- correspondiente al código, comenzó la secuencia en 1.

-- Ingresamos más registros:

```
insert into libros (titulo,autor,editorial)  
values('Martin Fierro','Jose Hernandez','Emece');  
  
insert into libros (titulo,autor,editorial)  
values('Aprenda PHP','Mario Molina','Emece');  
  
insert into libros (titulo,autor,editorial)  
values('Cervantes y el quijote','Borges','Paidos');  
  
insert into libros (titulo,autor,editorial)  
values('Matematica estas ahi', 'Paenza', 'Paidos');
```

-- Seleccionamos todos los registros:

```
select codigo,titulo,autor,editorial from libros;
```

-- Vemos que el código, dato que no ingresamos,

-- se cargó automáticamente siguiendo la secuencia de autoincremento.

-- Ahora borramos el libro con código 1:

```
delete from libros where codigo=1;
```

-- Mostramos todos los registros:

```
select * from libros;
```

-- Insertamos un nuevo libro e indicamos el valor que debe tomar el campo serial:

```
insert into libros (codigo,titulo,autor,editorial)
```

```
values(1,'Aprender Python', 'Rodriguez Luis', 'Paidos'); select * from libros;
```

-- Luego si insertamos otro registro sin indicar el valor del campo serial el valor

-- generado es el siguiente del último generado:

```
insert into libros (titulo,autor,editorial)
```

```
values('Java Ya', 'Nelson', 'Paidos');
```

```
select * from libros;
```

La ejecución de parte de este lote de comandos SQL genera una salida similar a:

The screenshot shows the pgAdmin 4 interface. The top bar has tabs for Dashboard, Properties, SQL, Statistics, Dependencies, Dependents, and a Query tab. The title bar says "Query - bd1 on postgres@PostgreSQL 11". The main area contains a code editor with numbered lines of SQL code. Lines 20-22 are comments about serial fields. Lines 23-26 show a select statement and a note about the sequence starting at 1. Lines 27-35 show multiple insert statements into the libros table. Lines 36-39 show another select statement and a note about auto-incrementing the code field. The bottom part of the interface shows a "Data Output" tab with a results grid. The grid has columns: codigo (integer), titulo (character varying(30)), autor (character varying(30)), and editorial (character varying(15)). There are 5 rows of data corresponding to the inserts.

	codigo	titulo	autor	editorial
	integer	character varying(30)	character varying(30)	character varying(15)
1	1	El aleph	Borges	Planeta
2	2	Martin Fierro	Jose Hernandez	Emece
3	3	Aprenda PHP	Mario Molina	Emece
4	4	Cervantes y el quijote	Borges	Paidos
5	5	Matematica estas ahi	Paenza	Paidos

14 - Comando truncate table

Aprendimos que para borrar todos los registro de una tabla se usa "delete" sin condición "where".

También podemos eliminar todos los registros de una tabla con "truncate table". Por ejemplo, queremos vaciar la tabla "libros", usamos:

```
truncate table libros;
```

La sentencia "truncate table" vacía la tabla (elimina todos los registros) y vuelve a crear la tabla con la misma estructura.

La diferencia con "drop table" es que esta sentencia borra la tabla, "truncate table" la vacía.

La diferencia con "delete" es la velocidad, es más rápido "truncate table" que "delete" (se nota cuando la cantidad de registros es muy grande) ya que éste borra los registros uno a uno.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
codigo serial,
```

```
titulo varchar(30),
```

```
autor varchar(30),
```

```
editorial varchar(15),
```

```
primary key (codigo)
```

```
);
```

```
insert into libros (titulo,autor,editorial)
```

```
values('Martin Fierro','Jose Hernandez','Planeta');
```

```
insert into libros (titulo,autor,editorial)
```

```
values('Aprenda PHP','Mario Molina','Emece');
```

```
insert into libros (titulo,autor,editorial)
```

```
values('Cervantes y el quijote','Borges','Paidos');
```

```
insert into libros (titulo,autor,editorial)
```

```
values('Matematica estas ahi', 'Paenza', 'Paidos');
```

```
insert into libros (titulo,autor,editorial)
```

```
values('El aleph', 'Borges', 'Emece');
```

-- Eliminemos todos los registros con "delete":

```
delete from libros;
```

-- Veamos el resultado:

```
select * from libros;
```

-- La tabla ya no contiene registros.

-- Ingresamos un nuevo registro:

```
insert into libros (titulo,autor,editorial)
```

```
values('Antología poetica', 'Borges', 'Emece');
```

-- Veamos el resultado:

```
select * from libros;
```

-- Para el campo "codigo" se guardó el valor 6 porque el valor

-- más alto de ese campo, antes de eliminar todos los registros era "5".

-- Ahora vaciemos la tabla:

```
truncate table libros;
```

-- Veamos qué sucede si ingresamos otro registro sin valor para el código:

```
insert into libros (titulo,autor,editorial)
```

```
values('Antología poetica', 'Borges', 'Emece');
```

-- Vemos que la secuencia de "codigo" continúa.

-- Ejecutamos entonces:

```
select * from libros;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows the pgAdmin 4 interface. The top bar has tabs for Dashboard, Properties, SQL, Statistics, Dependencies, Dependents, and a selected Query tab. Below the tabs is a toolbar with icons for file operations like Open, Save, and Print. A dropdown menu shows 'No limit' and other options. The main area contains a code editor with numbered lines 36 to 47. Lines 36-37 show a comment about the 'codigo' field. Line 38 starts a 'truncate' command. Lines 41-44 show an 'insert' command with a value for 'codigo'. Line 46 starts a 'select' command. Below the code editor is a results grid with tabs for Data Output, Explain, Messages, Notifications, and Query History. The Data Output tab is selected, showing a table with four columns: 'codigo' (integer), 'titulo' (character varying(30)), 'autor' (character varying(30)), and 'editorial' (character varying(15)). There is one row with values: 7, 'Antología poética', 'Borges', and 'Emece'.

	codigo	titulo	autor	editorial
	integer	character varying(30)	character varying(30)	character varying(15)
1	7	Antología poética	Borges	Emece

15 - Tipo de dato texto

Ya explicamos que al crear una tabla debemos elegir la estructura adecuada, esto es, definir los campos y sus tipos más precisos, según el caso.

Para almacenar TEXTO usamos cadenas de caracteres.

Las cadenas se colocan entre comillas simples.

Podemos almacenar letras, símbolos y dígitos con los que no se realizan operaciones matemáticas, por ejemplo, códigos de identificación, números de documentos, números telefónicos.

Tenemos los siguientes tipos:

varchar(x): define una cadena de caracteres de longitud variable en la cual determinamos el máximo de caracteres con el argumento "x" que va entre paréntesis.

. Su rango va de 1 a 10485760 caracteres.

char(x): define una cadena de longitud fija determinada por el argumento "x". Su rango es de 1 a 10485760 caracteres.

Si la longitud es invariable, es conveniente utilizar el tipo char; caso contrario, el tipo varchar. "char" viene de character, que significa carácter en inglés.

text: define una cadena de longitud variable, podemos almacenar una cadena de hasta 1GB (podemos utilizar las palabras claves character varying en lugar de text).

Si intentamos almacenar en un campo una cadena de caracteres de mayor longitud que la definida, aparece un mensaje indicando tal situación y la sentencia no se ejecuta.

Por ejemplo, si definimos un campo de tipo varchar(10) y le asignamos la cadena 'Aprenda PHP' (11 caracteres), aparece un mensaje y la sentencia no se ejecuta.

Si ingresamos un valor numérico (omitiendo las comillas), lo convierte a cadena y lo ingresa como tal.

Por ejemplo, si en un campo definido como varchar(5) ingresamos el valor 12345, lo toma como si hubiésemos tipeado '12345', igualmente, si ingresamos el valor 23.56, lo convierte a '23.56'. Si el valor numérico, al ser convertido a cadena supera la longitud definida, aparece un mensaje de error y la sentencia no se ejecuta.

Es importante elegir el tipo de dato adecuado según el caso, el más preciso.

Para almacenar cadenas que varían en su longitud, es decir, no todos los registros tendrán la misma longitud en un campo determinado, se emplea "varchar" en lugar de "char".

Por ejemplo, en campos que guardamos nombres y apellidos, no todos los nombres y apellidos tienen la misma longitud.

Para almacenar cadenas que no varían en su longitud, es decir, todos los registros tendrán la misma longitud en un campo determinado, se emplea "char".

Por ejemplo, definimos un campo "codigo" que constará de 5 caracteres, todos los registros tendrán un código de 5 caracteres, ni más ni menos.

Con PostgreSQL podemos utilizar como sinónimos las palabras claves 'character varying(n)' en lugar de 'varchar(n)', igualmente la palabra 'character(n)' remplazando a 'char(n)'.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists visitantes;
```

```
create table visitantes(
```

```
    nombre varchar(30),
```

```
    edad integer,
```

```
    sexo char(1),
```

```
    domicilio varchar(30),
```

```
    ciudad varchar(20),
```

```
    telefono varchar(11)
```

```
);
```

-- Intentamos ingresar una cadena de mayor longitud que la definida:

```
insert into visitantes (nombre,edad,sexo,domicilio,ciudad,telefono)
```

```
values ('Juan Juarez',32,'masc','Avellaneda 789','Cordoba','4234567');
```

-- aparece un mensaje de error y la sentencia no se ejecuta

-- Ingresamos un número telefónico olvidando las comillas, es decir,

-- como un valor numérico:

```
insert into visitantes (nombre,edad,sexo,domicilio,ciudad,telefono)
```

```
values ('Marcela Morales',43,'f','Colon 456','Cordoba',4567890);
```

-- lo convierte a cadena, veámoslo:

```
select * from visitantes;
```

-- Ahora borramos la tabla y la creamos utilizando como tipo de campo

-- los alias existentes para los tipos de datos varchar y char:

```
drop table visitantes;
```

```
create table visitantes(
```

```
    nombre character varying(30),
```

```
    edad integer,
```

```
    sexo character(1),
```

```
    domicilio character varying(30),
```

```
    ciudad character varying(20),
```

```
    telefono character varying(11)
```

```
);
```

-- Insertamos un registro:

```
insert into visitantes (nombre,edad,sexo,domicilio,ciudad,telefono)
```

```
values ('Marcela Morales',43,'f','Colon 456','Cordoba',4567890);
```

-- Mostramos el registro cargado:

```
select * from visitantes;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL 11
drop table if exists visitantes;
create table visitantes(
    nombre varchar(30),
    edad integer,
    sexo char(1),
    domicilio varchar(30),
    ciudad varchar(20),
    telefono varchar(11)
);
-- Intentamos ingresar una cadena de mayor longitud que la definida:
insert into visitantes (nombre,edad,sexo,domicilio,ciudad,telefono)
values ('Juan Juarez',32,'masc','Avellaneda 789','Cordoba','4234567');
-- aparece un mensaje de error y la sentencia no se ejecuta
```

Data Output Explain Messages Notifications Query History

ERROR: el valor es demasiado largo para el tipo character(1)
SQL state: 22001

16 - Tipo de dato numérico

Ya explicamos que al crear una tabla debemos elegir la estructura adecuada, esto es, definir los campos y sus tipos más precisos, según el caso.

Para almacenar valores NUMERICOS PostgreSQL dispone de varios tipos.

Para almacenar valores ENTEROS, por ejemplo, en campos que hacen referencia a cantidades, usamos:

- 1) int (integer o int4): su rango es de -2000000000 a 2000000000 aprox.
- 2) smallint (int2): Puede contener hasta 5 digitos. Su rango va desde -32000 hasta 32000 aprox.
- 3) bigint (int8): De -900000000000000000000000 hasta 900000000000000000000000 aprox.

Los campos de tipo serial : se almacenan en un campo de tipo int

y los bigserial : se almacenan en un campo de tipo bigint.

Para almacenar valores numéricos EXACTOS con decimales, especificando la cantidad de cifras a la izquierda y derecha del separador decimal, utilizamos:

- 4) decimal o numeric (t,d): Pueden tener hasta 1000 digitos, guarda un valor exacto. El primer argumento indica el total de dígitos y el segundo, la cantidad de decimales.
Por ejemplo, si queremos almacenar valores entre -99.99 y 99.99 debemos definir el campo como tipo "decimal(4,2)". Si no se indica el valor del segundo argumento, por defecto es "0". Por ejemplo, si definimos "decimal(4)" se pueden guardar valores entre -9999 y 9999.

El rango depende de los argumentos, también los bytes que ocupa.

Se utiliza el punto como separador de decimales.

Si ingresamos un valor con más decimales que los permitidos, redondea al más cercano; por ejemplo, si definimos "decimal(4,2)" e ingresamos el valor "12.686", guardará "12.69", redondeando hacia arriba; si ingresamos el valor "12.682", guardará "12.67", redondeando hacia abajo.

Para almacenar valores numéricos APROXIMADOS con decimales utilizamos:

- 5) float (real): De 1E-37 to 1E+37. Guarda valores aproximados.
- 6) double precision (float8): Desde 1E-307 to 1E+308. Guarda valores aproximados.

Para todos los tipos numéricos:

- si intentamos ingresar un valor fuera de rango, no lo permite.
- si ingresamos una cadena, PostgreSQL intenta convertirla a valor numérico, si dicha cadena consta solamente de dígitos, la conversión se realiza, luego verifica si está dentro del rango, si es así, la ingresa, sino, muestra un mensaje de error y no ejecuta la sentencia. Si la cadena contiene caracteres que PostgreSQL no puede convertir a valor numérico, muestra un mensaje de error y

la sentencia no se ejecuta.

Por ejemplo, definimos un campo de tipo decimal(5,2), si ingresamos la cadena '12.22', la convierte al valor numérico 12.22 y la ingresa; si intentamos ingresar la cadena '1234.56', la convierte al valor numérico 1234.56, pero como el máximo valor permitido es 999.99, muestra un mensaje indicando que está fuera de rango. Si intentamos ingresar el valor '12y.25', PostgreSQL no puede realizar la conversión y muestra un mensaje de error.

Es importante elegir el tipo de dato adecuado según el caso, el más preciso. Por ejemplo, si un campo numérico almacenará valores positivos menores a 255, el tipo "int" no es el más adecuado, conviene el tipo "smallint", de esta manera usamos el menor espacio de almacenamiento posible.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
    codigo serial,
    titulo varchar(40) not null,
    autor varchar(30),
    editorial varchar(15),
    precio decimal(10,2),
    cantidad smallint,
    primary key (codigo)
);
```

-- Analicemos la inserción de datos numéricos.

```
insert into libros (titulo,autor,editorial,precio,cantidad)
values('El aleph','Borges','Emece',25.6666, 260);
```

-- Se redondea el campo precio por el valor 25.67.

-- Intentamos ingresar un precio que supera el rango:

```
insert into libros (titulo,autor,editorial, precio, cantidad)
values('El aleph','Borges','Emece',120000000000.66, 260);
-- aparece un mensaje de error y la instrucción no se ejecuta.
```

-- Intentemos ingresar un valor mayor al permitido para el campo cantidad:

```
insert into libros (titulo,autor,editorial,precio,cantidad)  
values('El aleph','Borges','Emece',25000,100000);
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL query editor interface with the following content:

```
bd1 on postgres@PostgreSQL 11  
11 );  
12  
13 -- Analicemos la inserción de datos numéricos.  
14 insert into libros (titulo,autor,editorial,precio,cantidad)  
15 values('El aleph','Borges','Emece',25.6666, 260);  
16 -- Se redondea el campo precio por el valor 25.67.  
17  
18 -- Intentamos ingresar un precio que supera el rango:  
19 insert into libros (titulo,autor,editorial, precio, cantidad)  
20 values('El aleph','Borges','Emece',120000000000.66, 260);  
21 -- aparece un mensaje de error y la instrucción no se ejecuta.  
22  
23 -- Intentemos ingresar un valor mayor al permitido para el campo cantidad:  
24 insert into libros (titulo,autor,editorial,precio,cantidad)  
25 values('El aleph','Borges','Emece',25000,100000);  
26
```

Below the code area, there are tabs for Data Output, Explain, Messages, Notifications, and Query History. The Messages tab is active, displaying the following error message:

```
ERROR: desbordamiento de campo numeric  
DETAIL: Un campo con precisión 10, escala 2 debe redondear a un valor absoluto menor que 10^8.  
SQL state: 22003
```

17 - Tipo de dato fecha y hora

Ya explicamos que al crear una tabla debemos elegir la estructura adecuada, esto es, definir los campos y sus tipos más precisos, según el caso.

Para almacenar valores de tipo FECHA Y HORA PostgreSQL dispone de tres tipos:

- 1) date: almacena una fecha en el rango de 4713 antes de cristo hasta 32767 después de cristo.
- 2) time: Almacena la hora del día.
- 3) timestamp: Almacena la fecha y la hora del día.

Las fechas se ingresan entre comillas simples.

Para almacenar valores de tipo fecha se permiten como separadores "/", "-", "." entre otros.

PostgreSQL requiere que se ingrese la fecha con el formato aaaa/mm/dd:

```
insert into empleados values('Ana Gomez','22222222','2008/12/31');
```

PostgreSQL permite configurar el formato de ingreso de la fecha seteando la variable de entorno llamada DATESTYLE:

```
SET DATESTYLE TO 'European';
```

Con el ejemplo anterior luego podemos ingresar una fecha con el formato Europeo que es dd/mm/aaaa:

```
insert into empleados values('Pablo Rodriguez','11111111','31/12/2008');
```

Otros valores de seteo son:

ISO utiliza fechas y horas de estilo ISO 8601.

SQL utiliza fechas y horas de estilo Oracle/Ingres.

Postgres utiliza el formato tradicional de Postgres.

European utiliza dd/mm/aaaa para la representación numérica de las fechas.

NonEuropean utiliza mm/dd/aaaa para la representación numérica de las fechas.

German utiliza dd.mm.aaaa para la representación numérica de las fechas.

US igual que 'NonEuropean'

DEFAULT recupera los valores por defecto ('US,Postgres')

Para almacenar solo la hora debemos utilizar el tipo time. En un campo tipo time podemos almacenar un valor en el rango: 00:00:00.00 23:59:59.99.

```
insert into asistencia(fecha,hora) values ('2008/12/31','13:00:59');
```

Por último si queremos almacenar la fecha y la hora en un único campo debemos definirlo de tipo timestamp:

```
insert into asistencia (fechahora) values ('2008/12/31 13:00:00.59');
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists asistencia;
```

```
create table asistencia(
```

```
    dni char(8),
```

```
    fecha date,
```

```
    hora time,
```

```
    primary key (dni)
```

```
);
```

-- Ingresamos un registro:

```
insert into asistencia(dni,fecha,hora) values ('11111111','2008/12/31','13:00:59');
```

-- Mostramos el registro:

```
select * from asistencia;
```

-- Borramos la tabla:

```
drop table asistencia;
```

```
-- Creamos nuevamente la tabla pero definimos un solo campo para almacenar
```

```
-- la fecha y hora de ingreso del empleado:
```

```
create table asistencia(
```

```
    dni char(8),
```

```
    fechahora timestamp,
```

```
    primary key (dni)
```

```
);
```

```
-- Ingresamos un registro:
```

```
insert into asistencia (dni,fechahora) values ('11111111','2008/12/31 13:00:00.59');
```

```
-- Mostramos el registro:
```

```
select * from asistencia;
```

```
-- Cambiamos el seteo de fecha para ingresar con el formato dia/mes/año:
```

```
set datestyle to 'European';
```

```
-- Ingresamos un nuevo registro con el nuevo formato:
```

```
insert into asistencia (dni,fechahora) values ('22222222','21/12/2018 13:00:00.59');
```

```
-- Mostramos el registro:
```

```
select * from asistencia;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL:11
27
28 -- Ingresamos un registro:
29 insert into asistencia (dni,fechahora) values ('11111111','2008/12/31 13:00:00.59');
30
31 -- Mostramos el registro:
32 select * from asistencia;
33
34 -- Cambiamos el seteo de fecha para ingresar con el formato dia/mes/año:
35 set datestyle to 'European';
36
37 -- Ingresamos un nuevo registro con el nuevo formato:
38 insert into asistencia (dni,fechahora) values ('22222222','21/12/2018 13:00:00.59');
39
40 -- Mostramos el registro:
41 select * from asistencia;
42
```

Data Output Explain Messages Notifications Query History

dni	fechahora
11111111	timestamp without time zone
1 11111111	2008-12-31 13:00:00.59
2 22222222	2018-12-21 13:00:00.59

18 - Valores por defecto (default)

Hemos visto que si al insertar registros no se especifica un valor para un campo que admite valores nulos, se ingresa automáticamente "null" y si el campo está declarado serial o bigserial, se inserta el siguiente de la secuencia. A estos valores se les denomina valores por defecto o predeterminados.

Un valor por defecto se inserta cuando no está presente al ingresar un registro.

Para campos de cualquier tipo no declarados "not null", es decir, que admiten valores nulos, el valor por defecto es "null". Para campos declarados "not null", no existe valor por defecto, a menos que se declare explícitamente con la cláusula "default".

Para todos los tipos, excepto los declarados serial, se pueden explicitar valores por defecto con la cláusula "default".

Podemos establecer valores por defecto para los campos cuando creamos la tabla. Para ello utilizamos "default" al definir el campo. Por ejemplo, queremos que el valor por defecto del campo "autor" de la tabla "libros" sea "Desconocido" y el valor por defecto del campo "cantidad" sea "0":

```
create table libros(  
    codigo serial,  
    titulo varchar(40),  
    autor varchar(30) not null default 'Desconocido',  
    editorial varchar(20),  
    precio decimal(5,2),  
    cantidad smallint default 0,  
    primary key(codigo)  
);
```

Si al ingresar un nuevo registro omitimos los valores para el campo "autor" y "cantidad", PostgreSQL insertará los valores por defecto; el siguiente valor de la secuencia en "codigo", en "autor" colocará "Desconocido" y en cantidad "0".

Entonces, si al definir el campo explicitamos un valor mediante la cláusula "default", ése será el valor por defecto.

Ahora, al visualizar la estructura de la tabla podemos entender lo que informa la columna "column_default", muestra el valor por defecto del campo.

También se puede utilizar "default" para dar el valor por defecto a los campos en sentencias "insert", por ejemplo:

```
insert into libros (titulo,autor,precio,cantidad)
```

```
values ('El gato con botas',default,default,100);
```

Si todos los campos de una tabla tienen valores predeterminados (ya sea por ser "identity", permitir valores nulos o tener un valor por defecto), se puede ingresar un registro de la siguiente manera:

```
insert into libros default values;
```

La sentencia anterior almacenará un registro con los valores predeterminados para cada uno de sus campos.

Entonces, la cláusula "default" permite especificar el valor por defecto de un campo. Si no se explicita, el valor por defecto es "null", siempre que el campo no haya sido declarado "not null".

Los campos para los cuales no se ingresan valores en un "insert" tomarán los valores por defecto:

- si tiene el atributo "identity": el valor de inicio de la secuencia si es el primero o el siguiente valor de la secuencia, no admite cláusula "default";
- si permite valores nulos y no tiene cláusula "default", almacenará "null";
- si está declarado explícitamente "not null", no tiene valor "default" y no tiene el atributo "identity", no hay valor por defecto, así que causará un error y el "insert" no se ejecutará.
- si tiene cláusula "default" (admita o no valores nulos), el valor definido como predeterminado;
- para campos de tipo fecha y hora, si omitimos la parte de la fecha, el valor predeterminado para la fecha es "1900-01-01" y si omitimos la parte de la hora, "00:00:00".

Un campo sólo puede tener un valor por defecto. Una tabla puede tener todos sus campos con valores por defecto. Que un campo tenga valor por defecto no significa que no admita valores nulos, puede o no admitirlos.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
codigo serial,
```

```
titulo varchar(40),
```

```
autor varchar(30) not null default 'Desconocido',
```

```
editorial varchar(20),
```

```
precio decimal(5,2),
```

```
cantidad smallint default 0,
```

```
primary key(codigo)
);

insert into libros (titulo,editorial,precio)
values('Java en 10 minutos','Paidos',50.40);

-- PostgreSQL ingresará el registro con el siguiente valor de la secuencia
-- en "codigo", con el título, editorial y precio ingresados,
-- en "autor" colocará "Desconocido" y en cantidad "0":
```

```
select * from libros;
```

```
-- Si ingresamos un registro sin valor para el campo "precio",
-- que admite valores nulos, se ingresará "null" en ese campo:
```

```
insert into libros (titulo,editorial)
values('Aprenda PHP','Siglo XXI');
```

```
select * from libros;
```

```
-- Visualicemos la estructura de la tabla:
```

```
select *
from information_schema.columns
where table_name = 'libros';
```

```
-- La columna "column_default", muestra el valor por defecto de cada campo.
```

```
-- Podemos emplear "default" para dar el valor por defecto a algunos campos:
```

```
insert into libros (titulo,autor,precio,cantidad)
values ('El gato con botas',default,default,100);
select * from libros;
```

-- Como todos los campos de "libros" tienen valores predeterminados, podemos tipar:

```
insert into libros default values;
```

```
select * from libros;
```

-- La sentencia anterior almacenará un registro con los valores predeterminados

-- para cada uno de sus campos.

-- Que un campo tenga valor por defecto no significa que no admita valores nulos,

-- puede o no admitirlos. Podemos ingresar el valor "null" en el campo "cantidad":

```
insert into libros (titulo,autor,cantidad)
```

```
values ('Alicia en el pais de las maravillas','Lewis Carroll',null);
```

```
select * from libros;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL terminal window titled 'bd1 on postgres@PostgreSQL 11'. The command history (lines 36-51) includes several comments and SQL statements. Lines 36-37 show an insert with a default value for cantidad. Lines 39-44 show another insert with null for cantidad. Line 46-49 show a third insert with null for cantidad. Line 51 shows a select statement. A red box highlights the 'select * from libros;' command and its result. The result table has columns: codigo (integer), titulo (character varying(40)), autor (character varying(30)), editorial (character varying(20)), precio (numeric(5,2)), and cantidad (smallint). The data rows are:

	codigo	titulo	autor	editorial	precio	cantidad
1	1	Java en 10 minutos	Desconocido	Paidos	50.40	0
2	2	Aprenda PHP	Desconocido	Siglo XXI	[null]	0
3	3	El gato con botas	Desconocido	[null]	[null]	100
4	4	[null]	Desconocido	[null]	[null]	0
5	5	Alicia en el pais de las ma...	Lewis Carroll	[null]	[null]	[null]

19 - Columnas calculadas (operadores aritméticos y de concatenación)

Aprendimos que los operadores son símbolos que permiten realizar distintos tipos de operaciones.

Dijimos que PostgreSQL tiene 4 tipos de operadores: 1) relacionales o de comparación (los vimos), 2) lógicos (lo veremos más adelante, 3) aritméticos y 4) de concatenación.

Los operadores aritméticos permiten realizar cálculos con valores numéricos.

Son: multiplicación (*), división (/) y módulo (%) (el resto de dividir números enteros), suma (+) y resta (-).

Es posible obtener salidas en las cuales una columna sea el resultado de un cálculo y no un campo de una tabla.

Si queremos ver los títulos, precio y cantidad de cada libro escribimos la siguiente sentencia:

```
select titulo,precio,cantidad  
from libros;
```

Si queremos saber el monto total en dinero de un título podemos multiplicar el precio por la cantidad por cada título, pero también podemos hacer que PostgreSQL realice el cálculo y lo incluya en una columna extra en la salida:

```
select titulo, precio,cantidad,  
precio*cantidad  
from libros;
```

Si queremos saber el precio de cada libro con un 10% de descuento podemos incluir en la sentencia los siguientes cálculos:

```
select titulo,precio,  
precio-(precio*0.1)  
from libros;
```

También podemos actualizar los datos empleando operadores aritméticos:

```
update libros set precio=precio-(precio*0.1);
```

Todas las operaciones matemáticas retornan error si no se pueden ejecutar. Ejemplo:

```
select 5/0;
```

Los operadores de concatenación: permite concatenar cadenas, el más (||).

Para concatenar el título, el autor y la editorial de cada libro usamos el operador de concatenación ("||"):

```
select titulo||'-'||autor||'-'||editorial  
from libros;
```

Note que concatenamos además unos guiones para separar los campos.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(  
codigo serial,  
titulo varchar(40) not null,  
autor varchar(20) default 'Desconocido',  
editorial varchar(20),  
precio decimal(6,2),  
cantidad smallint default 0,  
primary key (codigo)  
);
```

-- Ingresamos algunos registros:

```
insert into libros (titulo,autor,editorial,precio)  
values('El aleph','Borges','Emece',25);  
  
insert into libros (titulo,autor,editorial,precio,cantidad)  
values('Java en 10 minutos','Mario Molina','Siglo XXI',50.40,100);  
  
insert into libros (titulo,autor,editorial,precio,cantidad)  
values('Alicia en el pais de las maravillas','Lewis Carroll','Emece',15,50);
```

-- Queremos saber el monto total en dinero de cada libro:

```
select titulo, precio,cantidad,
```

```
precio*cantidad
```

```
from libros;
```

--Queremos saber el precio de cada libro con un 10% de descuento:

```
select titulo,precio,
```

```
    precio-(precio*0.1)
```

```
from libros;
```

-- Actualizamos los precios con un 10% de descuento y vemos el resultado:

```
update libros set precio=precio-(precio*0.1);
```

```
select * from libros;
```

-- Queremos una columna con el título, el autor y la editorial de cada libro:

```
select titulo||'-'||autor||'-'||editorial
```

```
from libros;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bdf on postgres@PostgreSQL 11
14  insert into libros (titulo,autor,editorial,precio)
15    values('El aleph','Borges','Emece',25);
16  insert into libros (titulo,autor,editorial,precio,cantidad)
17    values('Java en 10 minutos','Mario Molina','Siglo XXI',50.40,100);
18  insert into libros (titulo,autor,editorial,precio,cantidad)
19    values('Alicia en el pais de las maravillas','Lewis Carroll','Emece',15,50);
20
21  -- Queremos saber el monto total en dinero de cada libro:
22  select titulo, precio,cantidad,
23    precio*cantidad
24  from libros;
25
26  --Queremos saber el precio de cada libro con un 10% de descuento:
27  select titulo,precio,
28    precio-(precio*0.1)
29  from libros;
```

Data Output Explain Messages Notifications Query History

titulo character varying (40)	precio numeric (6,2)	?column? numeric
1 El aleph	22.50	20.250
2 Java en 10 minutos	45.36	40.824
3 Alicia en el pais de las ma...	13.50	12.150

20 - Alias

Una manera de hacer más comprensible el resultado de una consulta consiste en cambiar los encabezados de las columnas.

Por ejemplo, tenemos la tabla "agenda" con un campo "nombre" (entre otros) en el cual se almacena el nombre y apellido de nuestros amigos; queremos que al mostrar la información de dicha tabla aparezca como encabezado del campo "nombre" el texto "nombre y apellido", para ello colocamos un alias de la siguiente manera:

```
select nombre as Nombreyapellido,  
domicilio,telefono  
from agenda;
```

Para reemplazar el nombre de un campo por otro, se coloca la palabra clave "as" seguido del texto del encabezado.

Se puede crear un alias para columnas calculadas.

Entonces, un "alias" se usa como nombre de un campo o de una expresión. En estos casos, son opcionales, sirven para hacer más comprensible el resultado; en otros casos, que veremos más adelante, son obligatorios.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists agenda;
```

```
create table agenda(
```

```
    nombre varchar(30),
```

```
    domicilio varchar(30),
```

```
    telefono varchar(11)
```

```
);
```

```
insert into agenda
```

```
values('Juan Perez','Avellaneda 908','4252525');
```

```
insert into agenda
```

```
values('Marta Lopez','Sucre 34','4556688');
```

```
insert into agenda
```

```
values('Carlos Garcia','Sarmiento 1258',null);
```

-- Mostramos la información con el encabezado "nombreyapellido" para el campo "nombre":

```
select nombre as nombreyapellido,  
domicilio,telefono  
from agenda;
```

-- Si tiene espacios en blanco el encabezado debe ir entre comillas dobles:

```
select nombre as "nombre y apellido",  
domicilio,telefono  
from agenda;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL 11  
1 drop table if exists agenda;  
2  
3 create table agenda(  
4     nombre varchar(30),  
5     domicilio varchar(30),  
6     telefono varchar(11)  
7 );  
8  
9 insert into agenda  
10    values('Juan Perez','Avellaneda 908','4252525');  
11 insert into agenda  
12    values('Marta Lopez','Sucre 34','4556688');  
13 insert into agenda  
14    values('Carlos Garcia','Sarmiento 1258',null);  
15  
16 -- Mostramos la información con el encabezado "nombreyapellido" para el campo "nombre":  
17 select nombre as nombreyapellido,  
18     domicilio,telefono  
19  from agenda;  
20  
21 -- Si tiene espacios en blanco el encabezado debe ir entre comillas dobles:  
22 select nombre as "nombre y apellido",  
23     domicilio,telefono  
24  from agenda;
```

	nombre y apellido	domicilio	telefono
1	Juan Perez	Avellaneda 908	4252525
2	Marta Lopez	Sucre 34	4556688
3	Carlos Garcia	Sarmiento 1258	[null]

21 - Funciones para el manejo de cadenas

PostgreSQL tiene funciones para trabajar con cadenas de caracteres. Estas son algunas:

`char_length(string)`: Retorna la longitud del texto. Ejemplo:

```
select char_length('Hola');
```

retorna un 4.

`upper(string)`: Retorna el texto convertido a mayúsculas. Ejemplo:

```
select upper('Hola');
```

retorna 'HOLA'.

`lower(string)`: Retorna el texto convertido a minúsculas. Ejemplo:

```
select lower('Hola');
```

retorna 'hola'.

`position(string in string)`: Retorna la posición de un string dentro de otro. Si no está contenido retorna un 0. Ejemplo:

```
select position('Mundo' in 'Hola Mundo');
```

retorna 6.

```
select position('MUNDO' in 'Hola Mundo');
```

retorna 0 (ya que no coinciden mayúsculas y minúsculas)

`substring(string [from int] [for int])`: Retorna un substring, le indicamos la posición inicial y la cantidad de caracteres a extraer desde dicha posición. Ejemplo:

```
select substring('Hola Mundo' from 1 for 2);
```

retorna 'Ho'.

```
select substring('Hola Mundo' from 6 for 5);
```

retorna 'Mundo'.

`trim([leading|trailing|both] [string] from string)`: Elimina caracteres del principio o final de un string. Por defecto elimina los espacios en blanco si no indicamos el carácter o string. Ejemplo:

```
select char_length(trim(' Hola Mundo '));
```

retorna un 10. Esto es debido a que primero se ejecuta la función trim que elimina los dos espacios iniciales y los dos finales.

```
select char_length(trim(leading ' ' from ' Hola Mundo '));
```

retorna un 12. Esto es debido a que indicamos que elimine los espacios en blanco de la cadena solo del comienzo (leading).

```
select trim(trailing '-' from '--Hola Mundo----');
```

retorna '--Hola Mundo'. Esto es debido a que indicamos que elimine los guiones del final del string.

`ltrim(string,string)`: Elimina los caracteres de la izquierda según el dato del segundo parámetro de la función. Ejemplo:

```
select char_length(ltrim(' Hola'));
```

retorna un 4.

```
select ltrim('---Hola','-'');
```

retorna 'Hola'.

`rtrim(string,string)`: Elimina los caracteres de la derecha según el dato del segundo parámetro de la función. Ejemplo:

```
select char_length(rtrim('Hola '));
```

retorna un 4.

```
select rtrim('Hola----','-'');
```

retorna un 'Hola'.

`substr(text,int[int])`: Retorna una subcadena a partir de la posición que le indicamos en el segundo parámetro hasta la posición indicada en el tercer parámetro. Ejemplo:

```
select substr('Hola Mundo',2,4);
```

retorna 'ola'.

```
select substr('Hola Mundo',2);
```

retorna 'ola Mundo' (si no indicamos el tercer parámetro retorna todo el string hasta el final)

lpad(text,int,text): Rellena de caracteres por la izquierda. El tamaño total de campo es indicado por el segundo parámetro y el texto a insertar se indica en el tercero. Ejemplo:

```
select lpad('Hola Mundo',20,'-');  
retorna '-----Hola Mundo'.
```

rpad(text,int,text): Rellena de caracteres por la derecha. El tamaño total de campo es indicado por el segundo parámetro y el texto a insertar se indica en el tercero. Ejemplo:

```
select rpad('Hola Mundo',20,'-');  
retorna 'Hola Mundo-----'.  
  
select rpad('Hola Mundo',20,'-*');  
retorna 'Hola Mundo-*-*-*-*'.
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(  
codigo serial,  
titulo varchar(40) not null,  
autor varchar(20) default 'Desconocido',  
editorial varchar(20),  
precio decimal(6,2),  
cantidad smallint default 0,  
primary key (codigo)  
);
```

```
insert into libros (titulo,autor,editorial,precio)  
values('El aleph','Borges','Emece',25);  
insert into libros (titulo,autor,editorial,precio,cantidad)
```

```
values('Java en 10 minutos','Mario Molina','Siglo XXI',50.40,100);

insert into libros (titulo,autor,editorial,precio,cantidad)

values('Alicia en el pais de las maravillas','Lewis Carroll','Emece',15,50);
```

-- Mostramos sólo los 12 primeros caracteres de los títulos de los libros y

-- sus autores, empleando la función "substring()":

```
select substring(titulo,1,12) as titulo
```

```
from libros;
```

-- Mostramos los títulos, autores y editoriales de todos libros,

-- al último campo lo queremos en mayúsculas:

```
select titulo, autor, upper(editorial)
```

```
from libros;
```

-- Mostrar todos los títulos de los libros rellenando con el

-- carácter '-' a la derecha.

```
select rpad(titulo,40,'-') from libros;
```

-- Imprimimos todo los libros que contienen la cadena 'en' en alguna

-- parte del título del libro.

```
select *
```

```
from libros
```

```
where position('en' in titulo)>0;
```

-- Imprimimos todos los libros que tienen un título con 10 o más caracteres:

```
select *
```

```
from libros
```

```
where char_length(titulo)>=10;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

Dashboard Properties SQL Statistics Dependencies Dependents Query - bd1 on postgres@PostgreSQL 11

No limit

bd1 on postgres@PostgreSQL 11

```
30 -- Mostrar todos los títulos de los libros rellenando con el
31 -- carácter '-' a la derecha.
32 select rpad(título,40,'-') from libros;
33
34 -- Imprimimos todo los libros que contienen la cadena 'en' en alguna
35 -- parte del título del libro.
36 select *
37   from libros
38  where position('en' in título)>0;
39
40 -- Imprimimos todos los libros que tienen un título con 10 o más caracteres:
41 select *
42   from libros
43  where char_length(título)>=10;
```

Data Output Explain Messages Notifications Query History

	codigo	título	autor	editorial	precio	cantidad
	integer	character varying (40)	character varying (20)	character varying (20)	numeric (6,2)	smallint
1	2	Java en 10 minutos	Mario Molina	Siglo XXI	50.40	100
2	3	Alicia en el país de las maravillas	Lewis Carroll	Emece	15.00	50

22 - Funciones matemáticas

Las funciones matemáticas realizan operaciones con expresiones numéricas y retornan un resultado, operan con tipos de datos numéricos.

PostgreSQL tiene algunas funciones para trabajar con números. Aquí presentamos algunas.

`abs(x)`: retorna el valor absoluto del argumento "x". Ejemplo:

```
select abs(-20);
```

retorna 20.

`cbrt(x)`: retorna la raíz cúbica del argumento "x". Ejemplo:

```
select cbrt(27);
```

retorna 3.

`ceiling(x)`: redondea hacia arriba el argumento "x". Ejemplo:

```
select ceiling(12.34);
```

retorna 13.

`floor(x)`: redondea hacia abajo el argumento "x". Ejemplo:

```
select floor(12.34);
```

retorna 12.

`power(x,y)`: retorna el valor de "x" elevado a la "y" potencia. Ejemplo:

```
select power(2,3);
```

retorna 8.

`round(numero)`: retorna un número redondeado al valor más próximo. Ejemplo:

```
select round(10.4);
```

retorna "10".

`sign(x)`: si el argumento es un valor positivo devuelve 1;-1 si es negativo y si es 0, 0. Ejemplo:

```
select sign(-23.4);
```

retorna "-1".

`sqrt(x)`: devuelve la raíz cuadrada del valor enviado como argumento. Ejemplo:

```
select sqrt(9);
```

retorna "3".

`mod(x,y)`: devuelve el resto de dividir x con respecto a y. Ejemplo:

```
select mod(11,2);
```

retorna "1".

`pi()`: devuelve el valor de pi. Ejemplo:

```
select pi();
```

retorna "3.14159265358979".

`random()`: devuelve un valor aleatorio entre 0 y 1 (sin incluirlos). Ejemplo:

```
select random();
```

retorna por ejemplo "0.895562474101578".

`trunc(x)`: Retorna la parte entera del parámetro. Ejemplo:

```
select trunc(34.7);
```

retorna "34".

`trunc(x,decimales)`: Retorna la parte entera del parámetro y la parte decimal truncando hasta el valor indicado en el segundo parámetro. Ejemplo:

```
select trunc(34.7777,2);
```

retorna "34.77".

$\sin(x)$: Retorna el valor del seno en radianes. Ejemplo:

```
select sin(0);
```

retorna "0".

$\cos(x)$: Retorna el valor del coseno en radianes. Ejemplo:

```
select cos(0);
```

retorna "1".

$\tan(x)$: Retorna el valor de la tangente en radianes. Ejemplo:

```
select tan(0);
```

retorna "0".

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
codigo serial,
```

```
titulo varchar(40) not null,
```

```
autor varchar(20) default 'Desconocido',
```

```
editorial varchar(20),
```

```
precio decimal(9,2),
```

```
primary key (codigo)
```

```
);
```

```
insert into libros (titulo,autor,editorial,precio)
```

```
values('El aleph','Borges','Emece',25.33);
```

```
insert into libros (titulo,autor,editorial,precio)
```

```
values('Java en 10 minutos','Mario Molina','Siglo XXI',50.65);
```

```

insert into libros (titulo,autor,editorial,precio)
values('Alicia en el pais de las maravillas','Lewis Carroll','Emece',19.95);

-- Vamos a mostrar los precios de los libros redondeando

-- el valor hacia abajo y hacia arriba:

select titulo,autor,precio,
       floor(precio) as abajo,
       ceiling(precio) as arriba
from libros;

```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows the pgAdmin 4 interface. The top bar has tabs for Dashboard, Properties, SQL, Statistics, Dependencies, Dependents, and a Query tab for 'Query - bd1 on postg'. Below the tabs is a toolbar with various icons. The main area is divided into two sections: a code editor and a results grid.

Code Editor Content:

```

11
12 insert into libros (titulo,autor,editorial,precio)
13   values('El aleph','Borges','Emece',25.33);
14 insert into libros (titulo,autor,editorial,precio)
15   values('Java en 10 minutos','Mario Molina','Siglo XXI',50.65);
16 insert into libros (titulo,autor,editorial,precio)
17   values('Alicia en el pais de las maravillas','Lewis Carroll','Emece',19.95);
18
19 -- Vamos a mostrar los precios de los libros redondeando
20 -- el valor hacia abajo y hacia arriba:
21 select titulo,autor,precio,
22       floor(precio) as abajo,
23       ceiling(precio) as arriba
24 from libros;
25

```

Results Grid:

	titulo	autor	precio	abajo	arriba
1	El aleph	Borges	25.33	25	26
2	Java en 10 minutos	Mario Molina	50.65	50	51
3	Alicia en el pais de las ma...	Lewis Carroll	19.95	19	20

23 - Funciones para el uso de fechas y horas

PostgreSQL ofrece algunas funciones para trabajar con fechas y horas. Estas son algunas:

- `current_date`: retorna la fecha actual. Ejemplo:

```
select current_date;
```

Retorna por ejemplo '2009-05-20'

- `current_time`: retorna la hora actual con la zona horaria. Ejemplo:

```
select current_time;
```

Retorna por ejemplo '18:33:06.074493+00'

- `current_timestamp`: retorna la fecha y la hora actual con la zona horaria. Ejemplo:

```
select current_timestamp;
```

Retorna por ejemplo '2009-05-20 18:34:16.63131+00'

- `extract(valor from timestamp)`: retorna una parte de la fecha u hora según le indiquemos antes del `from`, luego del `from` debemos indicar un campo o valor de tipo `timestamp` (o en su defecto anteceder la palabra clave `timestamp` para convertirlo). Ejemplo:

```
select extract(year from timestamp'2009-12-31 12:25:50');
```

Retorna el año '2009'

```
select extract(month from timestamp'2009-12-31 12:25:50');
```

Retorna el mes '12'

```
select extract(day from timestamp'2009-12-31 12:25:50');
```

Retorna el día '31'

```
select extract(hour from timestamp'2009-12-31 12:25:50');
```

Retorna la hora '12'

```
select extract(minute from timestamp'2009-12-31 12:25:50');
```

Retorna el minuto '25'

```
select extract(second from timestamp'2009-12-31 12:25:50');
```

Retorna el segundo '50'

```
select extract(century from timestamp'2009-12-31 12:25:50');
```

Retorna el siglo '21'

```
select extract(dow from timestamp'2009-12-31 12:25:50');
```

Retorna el día de la semana '4'

```
select extract(doy from timestamp'2009-12-31 12:25:50');
```

Retorna el día del año '365'

```
select extract(week from timestamp'2009-12-31 12:25:50');
```

Retorna el número de semana dentro del año '53'

```
select extract(quarter from timestamp'2009-12-31 12:25:50');
```

Retorna en qué cuarto del año se ubica la fecha '4'

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
    titulo varchar(40) not null,
```

```
    autor varchar(20) default 'Desconocido',
```

```
    editorial varchar(20),
```

```
    edicion timestamp,
```

```
    precio decimal(6,2)
```

```
);
```

```
insert into libros
```

```
values('El aleph','Borges','Emece','1980/10/10',25.33);
```

```
insert into libros
```

```
values('Java en 10 minutos','Mario Molina','Siglo XXI','2000/05/05',50.65);
```

```
insert into libros
```

```
values('Alicia en el país de las maravillas','Lewis Carroll','Emece','2000/08/09',19.95);
```

```
insert into libros  
values('Aprenda PHP','Mario Molina','Siglo XXI','2000/02/04',45);
```

-- Mostramos el título del libro y el año de edición:

```
select titulo, extract(year from edicion) from libros;
```

-- Mostramos el título del libro y el cuarto del año de edición:

```
select titulo, extract(quarter from edicion) from libros;
```

-- Muestre los títulos de los libros que se editaron el día 9,

--de cualquier mes de cualquier año:

```
select titulo from libros
```

```
where extract(day from edicion)=9;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL client interface with the following details:

- Toolbar:** Includes icons for file operations, search, and connection management.
- Query Bar:** Shows the connection name: `bd1 on postgres@PostgreSQL 11`.
- Code Area:** Displays the SQL script with line numbers (16 to 38). Lines 16-19 insert two books into the `libros` table. Lines 20-23 show how to query the year of publication. Lines 24-27 show how to query the quarter of publication. Lines 28-31 show how to query books published on day 9 of any month.
- Data Output Tab:** Active tab, showing the results of the final query (line 31). The table has one row with the title `Alicia en el país de las maravillas`.
- Other Tabs:** Explain, Messages, Notifications, Query History.

titulo
Alicia en el país de las maravillas

24 - Ordenar registros (order by)

Podemos ordenar el resultado de un "select" para que los registros se muestren ordenados por algún campo, para ello usamos la cláusula "order by".

La sintaxis básica es la siguiente:

```
select * from NOMBRETABLA  
order by CAMPO;
```

Por ejemplo, recuperamos los registros de la tabla "libros" ordenados por el título:

```
select * from libros  
order by titulo;
```

Aparecen los registros ordenados alfabéticamente por el campo especificado.

También podemos colocar el número de orden del campo por el que queremos que se ordene en lugar de su nombre, es decir, referenciar a los campos por su posición en la lista de selección. Por ejemplo, queremos el resultado del "select" ordenado por "precio":

```
select titulo,autor,precio  
from libros order by 3;
```

Por defecto, si no aclaramos en la sentencia, los ordena de manera ascendente (de menor a mayor).

Podemos ordenarlos de mayor a menor, para ello agregamos la palabra clave "desc":

```
select * libros  
order by editorial desc;
```

También podemos ordenar por varios campos, por ejemplo, por "titulo" y "editorial":

```
select * from libros  
order by titulo,editorial;
```

Incluso, podemos ordenar en distintos sentidos, por ejemplo, por "titulo" en sentido ascendente y "editorial" en sentido descendente:

```
select * from libros  
order by titulo asc, editorial desc;
```

Debe aclararse al lado de cada campo, pues estas palabras claves afectan al campo inmediatamente anterior.

Es posible ordenar por un campo que no se lista en la selección.

Se permite ordenar por valores calculados o expresiones.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
    codigo serial,
```

```
    titulo varchar(40) not null,
```

```
    autor varchar(20) default 'Desconocido',
```

```
    editorial varchar(20),
```

```
    precio decimal(6,2),
```

```
    primary key (codigo)
```

```
);
```

```
insert into libros (titulo,autor,editorial,precio)
```

```
values('El aleph','Borges','Emece',25.33);
```

```
insert into libros (titulo,autor,editorial,precio)
```

```
values('Java en 10 minutos','Mario Molina','Siglo XXI',50.65);
```

```
insert into libros (titulo,autor,editorial,precio)
```

```
values('Alicia en el pais de las maravillas','Lewis Carroll','Emece',19.95);
```

```
insert into libros (titulo,autor,editorial,precio)
```

```
values('Alicia en el pais de las maravillas','Lewis Carroll','Planeta',15);
```

```
-- Ordenamos los registros por el campo "precio", referenciando el campo por
```

```
-- su posición en la lista de selección:
```

```
select titulo,autor,precio
```

```
from libros order by 3;
```

```
-- Los ordenamos por "editorial", de mayor a menor empleando "desc":
```

```
select * from libros
```

```
order by editorial desc;
```

-- Ordenamos por dos campos:

```
select * from libros  
order by titulo,editorial;
```

-- Ordenamos en distintos sentidos:

```
select * from libros  
order by titulo asc, editorial desc;
```

-- Podemos ordenar por un campo que no se lista en la selección:

```
select titulo, autor  
from libros  
order by precio;
```

-- Está permitido ordenar por valores calculados:

```
select titulo, autor, editorial,  
precio+(precio*0.1) as preciocondescuento  
from libros  
order by 4;
```

-- Recuperamos los registros ordenados por el título:

```
select * from libros  
order by titulo;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```

bd1 on postgres@PostgreSQL 11
38 -- Podemos ordenar por un campo que no se lista en la selección:
39 select titulo, autor
40   from libros
41  order by precio;
42
43 -- Está permitido ordenar por valores calculados:
44 select titulo, autor, editorial,
45    precio+(precio*0.1) as preciocondescuento
46   from libros
47  order by 4;
48
49 -- Recuperaremos los registros ordenados por el título:
50 select * from libros
51 order by titulo;
52

```

Data Output Explain Messages Notifications Query History					
	codigo integer	titulo character varying (40)	autor character varying (20)	editorial character varying (20)	precio numeric (6,2)
1	3	Alicia en el país de las maravillas	Lewis Carroll	Emece	19.95
2	4	Alicia en el país de las maravillas	Lewis Carroll	Planeta	15.00
3	1	El aleph	Borges	Emece	25.33
4	2	Java en 10 minutos	Mario Molina	Siglo XXI	50.65

25 - Operadores lógicos (and - or - not)

Hasta el momento, hemos aprendido a establecer una condición con "where" utilizando operadores relacionales. Podemos establecer más de una condición con la cláusula "where", para ello aprenderemos los operadores lógicos.

Son los siguientes:

- and, significa "y",
- or, significa "y/o",
- not, significa "no", invierte el resultado
- (), paréntesis

Los operadores lógicos se usan para combinar condiciones.

Si queremos recuperar todos los libros cuyo autor sea igual a "Borges" y cuyo precio no supere los 20 euros, necesitamos 2 condiciones:

```
select * from libros  
where (autor='Borges') and  
(precio<=20);
```

Los registros recuperados en una sentencia que une 2 condiciones con el operador "and", cumplen con las 2 condiciones.

Queremos ver los libros cuyo autor sea "Borges" y/o cuya editorial sea "Planeta":

```
select * from libros  
where autor='Borges' or  
editorial='Planeta';
```

En la sentencia anterior usamos el operador "or"; indicamos que recupere los libros en los cuales el valor del campo "autor" sea "Borges" y/o el valor del campo "editorial" sea "Planeta", es decir, seleccionará los registros que cumplan con la primera condición, con la segunda condición o con ambas condiciones.

Los registros recuperados con una sentencia que une 2 condiciones con el operador "or", cumplen 1 de las condiciones o ambas.

Queremos recuperar los libros que NO cumplen la condición dada, por ejemplo, aquellos cuya editorial NO sea "Planeta":

```
select * from libros  
where not editorial='Planeta';
```

El operador "not" invierte el resultado de la condición a la cual antecede.

Los registros recuperados en una sentencia en la cual aparece el operador "not", no cumplen con la condición a la cual afecta el "NOT".

Los paréntesis se usan para encerrar condiciones, para que se evalúen como una sola expresión. Cuando explicitamos varias condiciones con diferentes operadores lógicos (combinamos "and", "or") permite establecer el orden de prioridad de la evaluación; además permite diferenciar las expresiones más claramente.

Por ejemplo, las siguientes expresiones devuelven un resultado diferente:

```
select * from libros  
where (autor='Borges') or  
(editorial='Paidos' and precio<20);
```

```
select * from libros  
where (autor='Borges' or editorial='Paidos') and  
(precio<20);
```

Si bien los paréntesis no son obligatorios en todos los casos, se recomienda utilizarlos para evitar confusiones.

El orden de prioridad de los operadores lógicos es el siguiente: "not" se aplica antes que "and" y "and" antes que "or", si no se especifica un orden de evaluación mediante el uso de paréntesis. El orden en el que se evalúan los operadores con igual nivel de precedencia es indefinido, por ello se recomienda usar los paréntesis.

Entonces, para establecer más de una condición en un "where" es necesario emplear operadores lógicos. "and" significa "y", indica que se cumplan ambas condiciones; "or" significa "y/o", indica que se cumpla una u otra condición (o ambas); "not" significa "no", indica que no se cumpla la condición especificada.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;  
  
create table libros(  
codigo serial,  
titulo varchar(40) not null,  
autor varchar(20) default 'Desconocido',  
editorial varchar(20),
```

```

precio decimal(6,2),
primary key(codigo)
);

insert into libros (titulo,autor,editorial,precio)
values('El aleph','Borges','Emece',15.90);

insert into libros (titulo,autor,editorial,precio)
values('Antología poética','Borges','Planeta',39.50);

insert into libros (titulo,autor,editorial,precio)
values('Java en 10 minutos','Mario Molina','Planeta',50.50);

insert into libros (titulo,autor,editorial,precio)
values('Alicia en el país de las maravillas','Lewis Carroll','Emece',19.90);

insert into libros (titulo,autor,editorial,precio)
values('Martin Fierro','Jose Hernandez','Emece',25.90);

insert into libros (titulo,autor,editorial,precio)
values('Martin Fierro','Jose Hernandez','Paidos',16.80);

insert into libros (titulo,autor,editorial,precio)
values('Aprenda PHP','Mario Molina','Emece',19.50);

insert into libros (titulo,autor,editorial,precio)
values('Cervantes y el Quijote','Borges','Paidos',18.40);

```

-- Seleccionamos los libros cuyo autor es "Borges" y/o cuya editorial es "Planeta":

```

select * from libros
where autor='Borges' or
editorial='Planeta';

```

-- Recuperamos los libros cuya editorial NO es "Planeta":

```

select * from libros
where not editorial='Planeta';

```

-- Veamos cómo el uso de paréntesis hace que PostgreSQL evalúe en forma diferente

-- ciertas consultas aparentemente iguales:

```
select * from libros
```

```
where (autor='Borges') or
```

```
(editorial='Paidos' and precio<20);
```

```
select * from libros
```

```
where (autor='Borges' or editorial='Paidos') and
```

```
(precio<20);
```

-- Recuperamos los libros cuyo autor sea igual a "Borges"

-- y cuyo precio no supere los 20 euros:

```
select * from libros
```

```
where (autor='Borges') and
```

```
(precio<=20);
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL 11
39 -- ciertas consultas aparentemente iguales:
40 select * from libros
41 where (autor='Borges') or
42 (editorial='Paidos' and precio<20);
43
44 select * from libros
45 where (autor='Borges' or editorial='Paidos') and
46 (precio<20);
47
48 -- Recuperamos los libros cuyo autor sea igual a "Borges"
49 -- y cuyo precio no supere los 20 pesos:
50 select * from libros
51 where (autor='Borges') and
52 (precio<=20);
```

Data Output	Explain	Messages	Notifications	Query History	
	codigo integer	titulo character varying (40)	autor character varying (20)	editorial character varying (20)	precio numeric (6,2)
1	1	El eleph	Borges	Emece	15.90
2	8	Cervantes y el quijote	Borges	Paidos	18.40

26 - Otros operadores relacionales (is null)

Hemos aprendido los operadores relacionales "=" (igual), "<>" (distinto), ">" (mayor), "<" (menor), ">=" (mayor o igual) y "<=" (menor o igual). Dijimos que no eran los únicos.

Existen otro operador relacional "is null".

Se emplea el operador "is null" para recuperar los registros en los cuales esté almacenado el valor "null" en un campo específico:

```
select * from libros
```

```
where editorial is null;
```

Para obtener los registros que no contiene "null", se puede emplear "is not null", esto mostrará los registros con valores conocidos.

Siempre que sea posible, emplee condiciones de búsqueda positivas ("is null"), evite las negativas ("is not null") porque con ellas se evalúan todos los registros y esto hace más lenta la recuperación de los datos.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
codigo serial,
```

```
titulo varchar(40) not null,
```

```
autor varchar(20) default 'Desconocido',
```

```
editorial varchar(20),
```

```
precio decimal(6,2),
```

```
primary key(codigo)
```

```
);
```

```
insert into libros(titulo,autor,editorial,precio)
```

```
values('El aleph','Borges','Emece',15.90);
```

```
insert into libros(titulo,autor,editorial,precio)
```

```
values('Cervantes y el quijote','Borges','Paidos',null);
```

```
insert into libros(titulo,autor,editorial,precio)
```

```
values('Alicia en el pais de las maravillas','Lewis Carroll',null,19.90);
```

```

insert into libros(titulo,autor,editorial,precio)
values('Martin Fierro','Jose Hernandez','Emece',25.90);

insert into libros (titulo,autor,precio)
values('Antología poética','Borges',25.50);

insert into libros (titulo,autor,precio)
values('Java en 10 minutos','Mario Molina',45.80);

insert into libros (titulo,autor)
values('Martin Fierro','Jose Hernandez');

insert into libros (titulo,autor)
values('Aprenda PHP','Mario Molina');

-- Seleccionamos los libros que no contiene "null" en "editorial":

select * from libros

where editorial is not null;

-- Recuperamos los registros en los cuales esté almacenado el valor "null"
-- en el campo "editorial":

select * from libros

where editorial is null;

```

La ejecución de este lote de comandos SQL genera una salida similar a:

```

bd1 on postgres@PostgreSQL 11
23: values('Java en 10 minutos','Mario Molina',45.80);
24: insert into libros (titulo,autor)
25: values('Martin Fierro','Jose Hernandez');
26: insert into libros (titulo,autor)
27: values('Aprenda PHP','Mario Molina');

28:
29: -- Seleccionamos los libros que no contiene "null" en "editorial":
30: select * from libros
31: where editorial is not null;
32:
33: -- Recuperamos los registros en los cuales esté almacenado el valor "null"
34: -- en el campo "editorial":
35: select * from libros
36: where editorial is null;
37:

```

	Data Output	Explain	Messages	Notifications	Query History																																			
	<table border="1"> <thead> <tr> <th>codigo</th> <th>titulo</th> <th>autor</th> <th>editorial</th> <th>precio</th> </tr> <tr> <th>integer</th> <th>character varying (40)</th> <th>character varying (20)</th> <th>character varying (20)</th> <th>numeric (6,2)</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>3 Alicia en el país de las maravillas</td> <td>Lewis Carroll</td> <td>[null]</td> <td>19.90</td> </tr> <tr> <td>2</td> <td>5 Antología poética</td> <td>Borges</td> <td>[null]</td> <td>25.50</td> </tr> <tr> <td>3</td> <td>6 Java en 10 minutos</td> <td>Mario Molina</td> <td>[null]</td> <td>45.80</td> </tr> <tr> <td>4</td> <td>7 Martin Fierro</td> <td>Jose Hernández</td> <td>[null]</td> <td>[null]</td> </tr> <tr> <td>5</td> <td>8 Aprenda PHP</td> <td>Mario Molina</td> <td>[null]</td> <td>[null]</td> </tr> </tbody> </table>	codigo	titulo	autor	editorial	precio	integer	character varying (40)	character varying (20)	character varying (20)	numeric (6,2)	1	3 Alicia en el país de las maravillas	Lewis Carroll	[null]	19.90	2	5 Antología poética	Borges	[null]	25.50	3	6 Java en 10 minutos	Mario Molina	[null]	45.80	4	7 Martin Fierro	Jose Hernández	[null]	[null]	5	8 Aprenda PHP	Mario Molina	[null]	[null]				
codigo	titulo	autor	editorial	precio																																				
integer	character varying (40)	character varying (20)	character varying (20)	numeric (6,2)																																				
1	3 Alicia en el país de las maravillas	Lewis Carroll	[null]	19.90																																				
2	5 Antología poética	Borges	[null]	25.50																																				
3	6 Java en 10 minutos	Mario Molina	[null]	45.80																																				
4	7 Martin Fierro	Jose Hernández	[null]	[null]																																				
5	8 Aprenda PHP	Mario Molina	[null]	[null]																																				

27 - Otros operadores relacionales (between)

Hemos visto los operadores relacionales: = (igual), <> (distinto), > (mayor), < (menor), >= (mayor o igual), <= (menor o igual), is null/is not null (si un valor es NULL o no).

Otro operador relacional es "between", trabajan con intervalos de valores.

Hasta ahora, para recuperar de la tabla "libros" los libros con precio mayor o igual a 20 y menor o igual a 40, usamos 2 condiciones unidas por el operador lógico "and":

```
select * from libros  
where precio>=20 and  
      precio<=40;
```

Podemos usar "between" y así simplificar la consulta:

```
select * from libros  
where precio between 20 and 40;
```

Averiguamos si el valor de un campo dado (precio) está entre los valores mínimo y máximo especificados (20 y 40 respectivamente).

"between" significa "entre". Trabaja con intervalo de valores.

Este operador se puede emplear con tipos de datos numéricos y tipos de datos fecha y hora (incluye sólo el valor mínimo).

No tiene en cuenta los valores "null".

Si agregamos el operador "not" antes de "between" el resultado se invierte, es decir, se recuperan los registros que están fuera del intervalo especificado. Por ejemplo, recuperamos los libros cuyo precio NO se encuentre entre 20 y 35, es decir, los menores a 15 y mayores a 25:

```
select * from libros  
where precio not between 20 and 35;
```

Siempre que sea posible, emplee condiciones de búsqueda positivas ("between"), evite las negativas ("not between") porque hace más lenta la recuperación de los datos.

Entonces, se puede usar el operador "between" para reducir las condiciones "where".

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
codigo serial,  
titulo varchar(40) not null,  
autor varchar(20) default 'Desconocido',  
editorial varchar(20),  
precio decimal(6,2),  
primary key(codigo)  
);
```

```
insert into libros(titulo,autor,editorial,precio)  
values('El aleph','Borges','Emece',15.90);  
insert into libros(titulo,autor,editorial,precio)  
values('Cervantes y el quijote','Borges','Paidos',null);  
insert into libros(titulo,autor,editorial,precio)  
values('Alicia en el pais de las maravillas','Lewis Carroll',null,19.90);  
insert into libros(titulo,autor,editorial,precio)  
values('Martin Fierro','Jose Hernandez','Emece',25.90);  
insert into libros (titulo,autor,precio)  
values('Antología poética','Borges',32);  
insert into libros (titulo,autor,precio)  
values('Java en 10 minutos','Mario Molina',45.80);  
insert into libros (titulo,autor,precio)  
values('Martin Fierro','Jose Hernandez',40);  
insert into libros (titulo,autor,precio)  
values('Aprenda PHP','Mario Molina',56.50);
```

```
-- Para seleccionar los libros cuyo precio NO esté entre un intervalo de valores
```

```
-- antecedemos "not" al "between":
```

```
select * from libros  
where precio not between 20 and 35;
```

-- Recuperamos los registros cuyo precio esté entre 20 y 40 empleando "between":

```
select * from libros
```

```
where precio between 20 and 40;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL:11
23   values('Java en 10 minutos','Mario Molina',45.80);
24   insert into libros (titulo,autor,precio)
25     values('Martin Fierro','Jose Hernandez',40);
26   insert into libros (titulo,autor,precio)
27     values('Aprenda PHP','Mario Molina',56.50);
28
29   -- Para seleccionar los libros cuyo precio NO esté entre un intervalo de valores
30   -- antecedemos "not" al "between":
31   select * from libros
32   where precio not between 20 and 35;
33
34   -- Recuperamos los registros cuyo precio esté entre 20 y 40 empleando "between":
35   select * from libros
36   where precio between 20 and 40;
37
```

Data Output				
Explain Messages Notifications Query History				
	codigo integer	titulo character varying (40)	autor character varying (20)	editorial character varying (20)
1	4	Martin Fierro	Jose Hernandez	Emece
2	5	Antologia poética	Borges	[null]
3	7	Martin Fierro	Jose Hernandez	[null]

28 - Otros operadores relacionales (in)

Se utiliza "in" para averiguar si el valor de un campo está incluido en una lista de valores especificada.

En la siguiente sentencia usamos "in" para averiguar si el valor del campo autor está incluido en la lista de valores especificada (en este caso, 2 cadenas).

Hasta ahora, para recuperar los libros cuyo autor sea 'Paenza' o 'Borges' usábamos 2 condiciones:

```
select * from libros  
where autor='Borges' or autor='Paenza';
```

Podemos usar "in" y simplificar la consulta:

```
select * from libros  
where autor in('Borges','Paenza');
```

Para recuperar los libros cuyo autor no sea 'Paenza' ni 'Borges' usábamos:

```
select * from libros  
where autor<>'Borges' and  
autor<>'Paenza';
```

También podemos usar "in" anteponiendo "not":

```
select * from libros  
where autor not in ('Borges','Paenza');
```

Empleando "in" averiguamos si el valor del campo está incluido en la lista de valores especificada; con "not" antecediendo la condición, invertimos el resultado, es decir, recuperamos los valores que no se encuentran (coinciden) con la lista de valores.

Los valores "null" no se consideran.

Recuerde: siempre que sea posible, emplee condiciones de búsqueda positivas ("in"), evite las negativas ("not in") porque con ellas se evalúan todos los registros y esto hace más lenta la recuperación de los datos.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```

codigo serial,
titulo varchar(40) not null,
autor varchar(20),
editorial varchar(20),
precio decimal(6,2),
primary key(codigo)

);

insert into libros(titulo,autor,editorial,precio)
values('El aleph','Borges','Emece',15.90);

insert into libros(titulo,autor,editorial,precio)
values('Cervantes y el quijote','Borges','Paidos',null);

insert into libros(titulo,autor,editorial,precio)
values('Alicia en el pais de las maravillas','Lewis Carroll',null,19.90);

insert into libros(titulo,autor,editorial,precio)
values('Matematica estas ahi','Paenza','Siglo XXI',15);

insert into libros (titulo,precio)
values('Antología poética',32);

insert into libros (titulo,autor,precio)
values('Martin Fierro','Jose Hernandez',40);

insert into libros (titulo,autor,precio)
values('Aprenda PHP','Mario Molina',56.50);

-- Recuperamos los libros cuyo autor es "Paenza" o "Borges":
select * from libros
where autor in('Borges','Paenza');

-- Recuperamos los libros cuyo autor NO es "Paenza" ni "Borges":
select * from libros

```

```
where autor not in ('Borges','Paenza');
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bdt@postgres@PostgreSQL:11
20  insert into libros (titulo,precio)
21    values('Antología poética',32);
22  insert into libros (titulo,autor,precio)
23    values('Martín Fierro','Jose Hernandez',40);
24  insert into libros (titulo,autor,precio)
25    values('Aprenda PHP','Mario Molina',56.50);
26
27  -- Recuperamos los libros cuyo autor es "Paenza" o "Borges":
28  select * from libros
29  where autor in('Borges','Paenza');
30
31  -- Recuperamos los libros cuyo autor NO es "Paenza" ni "Borges":
32  select * from libros
33  where autor not in ('Borges','Paenza');
```

Data Output					
	código integer	título character varying (40)	autor character varying (20)	editorial character varying (20)	precio numeric (6,2)
1	3	Alicia en el país de las maravillas	Lewis Carroll	[null]	19.90
2	6	Martín Fierro	Jose Hernandez	[null]	40.00
3	7	Aprenda PHP	Mario Molina	[null]	56.50

29 - Búsqueda de patrones (like - not like)

Existe un operador relacional que se usa para realizar comparaciones exclusivamente de cadenas, "like" y "not like".

Hemos realizado consultas utilizando operadores relacionales para comparar cadenas. Por ejemplo, sabemos recuperar los libros cuyo autor sea igual a la cadena "Borges":

```
select * from libros  
where autor='Borges';
```

El operador igual ("=") nos permite comparar cadenas de caracteres, pero al realizar la comparación, busca coincidencias de cadenas completas, realiza una búsqueda exacta.

Imaginemos que tenemos registrados estos 2 libros:

```
"El Aleph", "Borges";  
"Antología poética", "J.L. Borges";
```

Si queremos recuperar todos los libros de "Borges" y especificamos la siguiente condición:

```
select * from libros  
where autor='Borges';
```

sólo aparecerá el primer registro, ya que la cadena "Borges" no es igual a la cadena "J.L. Borges".

Esto sucede porque el operador "=" (igual), también el operador "<>" (distinto) comparan cadenas de caracteres completas. Para comparar porciones de cadenas utilizamos los operadores "like" y "not like".

Entonces, podemos comparar trozos de cadenas de caracteres para realizar consultas. Para recuperar todos los registros cuyo autor contenga la cadena "Borges" debemos tipar:

```
select * from libros  
where autor like "%Borges%";
```

El símbolo "%" (porcentaje) reemplaza cualquier cantidad de caracteres (incluyendo ningún carácter). Es un carácter comodín. "like" y "not like" son operadores de comparación que señalan igualdad o diferencia.

Para seleccionar todos los libros que comiencen con "M":

```
select * from libros  
where título like 'M%';
```

Note que el símbolo "%" ya no está al comienzo, con esto indicamos que el título debe tener como primera letra la "M" y luego, cualquier cantidad de caracteres.

Para seleccionar todos los libros que NO comiencen con "M":

```
select * from libros  
where titulo not like 'M%';
```

Así como "%" reemplaza cualquier cantidad de caracteres, el guión bajo "_" reemplaza un carácter, es otro carácter comodín. Por ejemplo, queremos ver los libros de "Lewis Carroll" pero no recordamos si se escribe "Carroll" o "Carrolt", entonces tipeamos esta condición:

```
select * from libros  
where autor like "%Carrol_";
```

"like" se emplea con tipos de datos char, varchar, date, time, timestamp. Si empleamos "like" con tipos de datos que no son caracteres utilizamos el comando 'cast' convirtiendo éste al tipo de dato. Por ejemplo, queremos buscar todos los libros cuyo precio se encuentre entre 10.00 y 19.99:

```
select titulo,precio from libros  
where cast(precio as varchar) like '1_.%';
```

Queremos los libros que NO incluyen centavos en sus precios:

```
select titulo,precio from libros  
where cast(precio as varchar) like '%.00';
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(  
codigo serial,  
titulo varchar(40) not null,  
autor varchar(20) default 'Desconocido',  
editorial varchar(20),  
precio decimal(6,2),  
primary key(codigo)  
);
```

```
insert into libros(titulo,autor,editorial,precio)  
values('El aleph','Borges','Emece',15.90);
```

```
insert into libros(titulo,autor,editorial,precio)
values('Antología poética','J. L. Borges','Planeta',null);

insert into libros(titulo,autor,editorial,precio)
values('Alicia en el pais de las maravillas','Lewis Carroll',null,19.90);

insert into libros(titulo,autor,editorial,precio)
values('Matematica estas ahi','Paenza','Siglo XXI',15);

insert into libros(titulo,autor,editorial,precio)
values('Martin Fierro','Jose Hernandez',default,40);

insert into libros(titulo,autor,editorial,precio)
values('Aprenda PHP','Mario Molina','Nuevo siglo',56.50);
```

-- Recuperamos todos los libros que contengan en el campo "autor" la cadena "Borges":

```
select * from libros
where autor like '%Borges%';
```

-- Seleccionamos los libros cuyos títulos comienzan con la letra "M":

```
select * from libros
where titulo like 'M%';
```

-- Seleccionamos todos los títulos que NO comienzan con "M":

```
select * from libros
where titulo not like 'M%';
```

-- Si queremos ver los libros de "Lewis Carroll" pero no recordamos si se escribe "Carroll"

-- o "Carrolt", podemos emplear el comodín "_" (guión bajo) y establecer la siguiente condición:

```
select * from libros
where autor like '%Carrol_';
```

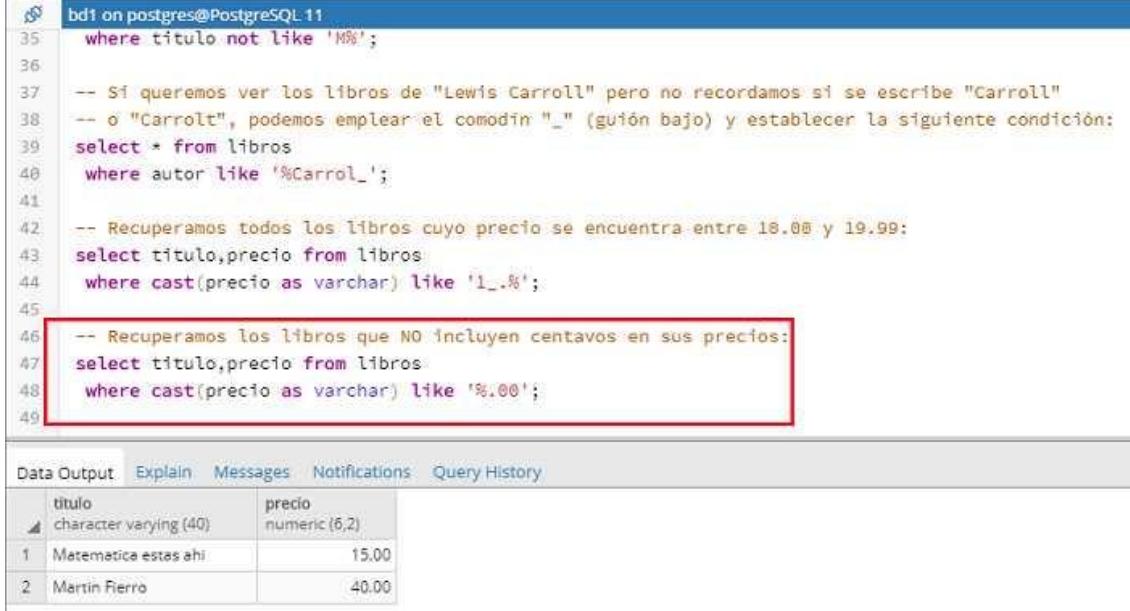
-- Recuperamos todos los libros cuyo precio se encuentra entre 10.00 y 19.99:

```
select titulo,precio from libros  
where cast(precio as varchar) like '1_.%';
```

-- Recuperamos los libros que NO incluyen centavos en sus precios:

```
select titulo,precio from libros  
where cast(precio as varchar) like '%.00';
```

La ejecución de este lote de comandos SQL genera una salida similar a:



```
bd1 on postgres@PostgreSQL 11  
35  where titulo not like '%.%';  
36  
37  -- Si queremos ver los libros de "Lewis Carroll" pero no recordamos si se escribe "Carroll"  
38  -- o "Carrott", podemos emplear el comodín "_" (guion bajo) y establecer la siguiente condición:  
39  select * from libros  
40  where autor like '%Carrol_';  
41  
42  -- Recuperamos todos los libros cuyo precio se encuentra entre 18.00 y 19.99:  
43  select titulo,precio from libros  
44  where cast(precio as varchar) like '1_.%';  
45  
46  -- Recuperamos los libros que NO incluyen centavos en sus precios:  
47  select titulo,precio from libros  
48  where cast(precio as varchar) like '%.00';
```

Data Output	
titulo	precio
character varying (40)	numeric (6,2)
1 Matematica estas ahí	15.00
2 Martín Fierro	40.00

30 - Contar registros (count)

Existen en PostgreSQL funciones que nos permiten contar registros, calcular sumas, promedios, obtener valores máximos y mínimos. Estas funciones se denominan funciones de agregado y operan sobre un conjunto de valores (registros), no con datos individuales y devuelven un único valor.

Imaginemos que nuestra tabla "libros" contiene muchos registros. Para averiguar la cantidad sin necesidad de contarlos manualmente usamos la función "count()":

```
select count(*)
```

```
from libros;
```

La función "count()" cuenta la cantidad de registros de una tabla, incluyendo los que tienen valor nulo.

También podemos utilizar esta función junto con la cláusula "where" para una consulta más específica. Queremos saber la cantidad de libros de la editorial "Planeta":

```
select count(*)
```

```
from libros
```

```
where editorial='Planeta';
```

Para contar los registros que tienen precio (sin tener en cuenta los que tienen valor nulo), usamos la función "count()" y en los paréntesis colocamos el nombre del campo que necesitamos contar:

```
select count(precio)
```

```
from libros;
```

Note que "count(*)" retorna la cantidad de registros de una tabla (incluyendo los que tienen valor "null") mientras que "count(precio)" retorna la cantidad de registros en los cuales el campo "precio" no es nulo. No es lo mismo. "count(*)" cuenta registros, si en lugar de un asterisco colocamos como argumento el nombre de un campo, se contabilizan los registros cuyo valor en ese campo NO es nulo.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
codigo serial,
```

```
titulo varchar(40) not null,
```

```

autor varchar(20) default 'Desconocido',
editorial varchar(20),
precio decimal(6,2),
primary key(codigo)

);

insert into libros(titulo,autor,editorial,precio)
values('El aleph','Borges','Emece',15.90);

insert into libros(titulo,autor,editorial,precio)
values('Antología poética','J. L. Borges','Planeta',null);

insert into libros(titulo,autor,editorial,precio)
values('Alicia en el país de las maravillas','Lewis Carroll',null,19.90);

insert into libros(titulo,autor,editorial,precio)
values('Matemática estas ahí','Paenza','Siglo XXI',15);

insert into libros(titulo,autor,editorial,precio)
values('Martin Fierro','Jose Hernandez',default,40);

insert into libros(titulo,autor,editorial,precio)
values('Aprenda PHP','Mario Molina','Nuevo siglo',null);

insert into libros(titulo,autor,editorial,precio)
values('Uno','Richard Bach','Planeta',20);

-- Averiguemos la cantidad de libros usando la función "count()":
select count(*)
from libros;

-- Note que incluye todos los libros aunque tengan valor nulo en algún campo.

-- Contamos los libros de editorial "Planeta":
select count(*)
from libros

```

```
where editorial='Planeta';

-- Contamos los registros que tienen precio (sin tener en cuenta los que
-- tienen valor nulo), usando la función "count(precio)":

select count(precio)
from libros;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL terminal window titled 'bd1 on postgres@PostgreSQL:11'. The code area contains several SQL queries numbered 27 to 41. A red box highlights the following code block:

```
-- Contamos los registros que tienen precio (sin tener en cuenta los que
-- tienen valor nulo), usando la función "count(precio)":  
select count(precio)  
from libros;
```

Below the code, there is a table labeled 'Data Output' with one row of data:

count	bigint
1	5

31 - Funciones de agrupamiento (count - sum - min - max - avg)

Hemos visto que PostgreSQL tiene funciones que nos permiten contar registros, calcular sumas, promedios, obtener valores máximos y mínimos, las funciones de agregado.

Ya hemos aprendido una de ellas, "count()", veamos otras.

Se pueden usar en una instrucción "select" y combinarlas con la cláusula "group by".

Todas estas funciones retornan "null" si ningún registro cumple con la condición del "where", excepto "count" que en tal caso retorna cero.

El tipo de dato del campo determina las funciones que se pueden emplear con ellas.

Las relaciones entre las funciones de agrupamiento y los tipos de datos es la siguiente:

- count: se puede emplear con cualquier tipo de dato.
- min y max: con cualquier tipo de dato.
- sum y avg: sólo en campos de tipo numérico.

La función "sum()" retorna la suma de los valores que contiene el campo especificado. Si queremos saber la cantidad total de libros que tenemos disponibles para la venta, debemos sumar todos los valores del campo "cantidad":

```
select sum(cantidad)
```

```
from libros;
```

Para averiguar el valor máximo o mínimo de un campo usamos las funciones "max()" y "min()" respectivamente.

Queremos saber cuál es el mayor precio de todos los libros:

```
select max(precio)
```

```
from libros;
```

Entonces, dentro del paréntesis de la función colocamos el nombre del campo del cuál queremos el máximo valor.

La función "avg()" retorna el valor promedio de los valores del campo especificado. Queremos saber el promedio del precio de los libros referentes a "PHP":

```
select avg(precio)
```

```
from libros
```

```
where titulo like '%PHP%';
```

Ahora podemos entender porque estas funciones se denominan "funciones de agrupamiento", porque operan sobre conjuntos de registros, no con datos individuales.

Tratamiento de los valores nulos:

Si realiza una consulta con la función "count" de un campo que contiene 18 registros, 2 de los cuales contienen valor nulo, el resultado devuelve un total de 16 filas porque no considera aquellos con valor nulo.

Todas las funciones de agregado, excepto "count(*)", excluye los valores nulos de los campos. "count(*)" cuenta todos los registros, incluidos los que contienen "null".

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
    codigo serial,
```

```
    titulo varchar(40) not null,
```

```
    autor varchar(30) default 'Desconocido',
```

```
    editorial varchar(15),
```

```
    precio decimal(5,2),
```

```
    cantidad smallint,
```

```
    primary key(codigo)
```

```
);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('El aleph','Borges','Planeta',15,null);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('Martin Fierro','Jose Hernandez','Emece',22.20,200);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('Antologia poetica','J.L. Borges','Planeta',null,150);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('Aprenda PHP','Mario Molina','Emece',18.20,null);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('Cervantes y el quijote','Bioy Casares- J.L. Borges','Paidos',null,100);

insert into libros(titulo,autor,editorial,precio,cantidad)

values('Manual de PHP', 'J.C. Paez', 'Siglo XXI',31.80,120);

insert into libros(titulo,autor,editorial,precio,cantidad)

values('Harry Potter y la piedra filosofal','J.K. Rowling',default,45.00,90);

insert into libros(titulo,autor,editorial,precio,cantidad)

values('Harry Potter y la camara secreta','J.K. Rowling','Emece',46.00,100);

insert into libros (titulo,autor,cantidad)

values('Alicia en el pais de las maravillas','Lewis Carroll',220);

insert into libros (titulo,autor,cantidad)

values('PHP de la A a la Z',default,0);
```

-- Para conocer la cantidad total de libros, sumamos las cantidades de cada uno:

```
select sum(cantidad)

from libros;
```

-- Queremos saber cuántos libros tenemos de la editorial "Emece":

```
select sum(cantidad)

from libros

where editorial='Emece';
```

-- Queremos saber cuál es el libro más costoso; usamos la función "max()":

```
select max(precio)

from libros;
```

-- Para conocer el precio mínimo de los libros de "Rowling" tipeamos:

```
select min(precio)

from libros

where autor like '%Rowling%';
```

-- Queremos saber el promedio del precio de los libros referentes a "PHP":

```
select avg(precio)  
from libros  
where titulo like '%PHP%';
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL terminal window titled "bd1 on postgres@PostgreSQL 11". The query editor contains the following code:

```
42:  
43: -- Queremos saber cuál es el libro más costoso; usamos la función "max()":  
44: select max(precio)  
45:   from libros;  
46:  
47: -- Para conocer el precio mínimo de los libros de "Rowling" tipeamos:  
48: select min(precio)  
49:   from libros  
50:  where autor like '%Rowling%';  
51:  
52: -- Queremos saber el promedio del precio de los libros referentes a "PHP":  
53: select avg(precio)  
54:   from libros  
55:  where titulo like '%PHP%';  
56: |
```

Below the code, there is a "Data Output" tab with a single row of results:

avg
numeric
1 25.0000000000000000000000

32 - Agrupar registros (group by)

Hemos aprendido que las funciones de agregado permiten realizar varios cálculos operando con conjuntos de registros.

Las funciones de agregado solas producen un valor de resumen para todos los registros de un campo. Podemos generar valores de resumen para un solo campo, combinando las funciones de agregado con la cláusula "group by", que agrupa registros para consultas detalladas.

Queremos saber la cantidad de libros de cada editorial, podemos tipar la siguiente sentencia:

```
select count(*) from libros
```

```
where editorial='Planeta';
```

y repetirla con cada valor de "editorial":

```
select count(*) from libros
```

```
where editorial='Emece';
```

```
select count(*) from libros
```

```
where editorial='Paidos';
```

```
...
```

Pero hay otra manera, utilizando la cláusula "group by":

```
select editorial, count(*)
```

```
from libros
```

```
group by editorial;
```

La instrucción anterior solicita que muestre el nombre de la editorial y cuente la cantidad agrupando los registros por el campo "editorial". Como resultado aparecen los nombres de las editoriales y la cantidad de registros para cada valor del campo.

Los valores nulos se procesan como otro grupo.

Entonces, para saber la cantidad de libros que tenemos de cada editorial, utilizamos la función "count()", agregamos "group by" (que agrupa registros) y el campo por el que deseamos que se realice el agrupamiento, también colocamos el nombre del campo a recuperar; la sintaxis básica es la siguiente:

```
select CAMPO, FUNCIONDEAGREGADO
```

```
from NOMBRERATABLA
```

```
group by CAMPO;
```

También se puede agrupar por más de un campo, en tal caso, luego del "group by" se listan los campos, separados por comas. Todos los campos que se especifican en la cláusula "group by" deben estar en la lista de selección.

```
select CAMPO1, CAMPO2, FUNCIONDEAGREGADO
```

```
from NOMBRERATABLA
```

```
group by CAMPO1,CAMPO2;
```

Para obtener la cantidad libros con precio no nulo, de cada editorial utilizamos la función "count()" enviándole como argumento el campo "precio", agregamos "group by" y el campo por el que deseamos que se realice el agrupamiento (editorial):

```
select editorial, count(precio)
```

```
from libros
```

```
group by editorial;
```

Como resultado aparecen los nombres de las editoriales y la cantidad de registros de cada una, sin contar los que tienen precio nulo.

Recuerde la diferencia de los valores que retorna la función "count()" cuando enviamos como argumento un asterisco o el nombre de un campo: en el primer caso cuenta todos los registros incluyendo los que tienen valor nulo, en el segundo, los registros en los cuales el campo especificado es no nulo.

Para conocer el total en dinero de los libros agrupados por editorial:

```
select editorial, sum(precio)
```

```
from libros
```

```
group by editorial;
```

Para saber el máximo y mínimo valor de los libros agrupados por editorial:

```
select editorial,
```

```
max(precio) as mayor,
```

```
min(precio) as menor
```

```
from libros
```

```
group by editorial;
```

Para calcular el promedio del valor de los libros agrupados por editorial:

```
select editorial, avg(precio)
```

```
from libros
```

```
group by editorial;
```

Es posible limitar la consulta con "where".

Si incluye una cláusula "where", sólo se agrupan los registros que cumplen las condiciones.

Vamos a contar y agrupar por editorial considerando solamente los libros cuyo precio sea menor a 30 euros:

```
select editorial, count(*)
```

```
from libros
```

```
where precio<30
```

```
group by editorial;
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
codigo serial,
```

```
titulo varchar(40),
```

```
autor varchar(30),
```

```
editorial varchar(15),
```

```
precio decimal(5,2),
```

```
cantidad smallint,
```

```
primary key(codigo)
```

```
);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('El aleph','Borges','Planeta',15,null);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('Martin Fierro','Jose Hernandez','Emece',22.20,200);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('Antologia poetica','J.L. Borges','Planeta',null,150);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
values('Aprenda PHP','Mario Molina','Emece',18.20,null);

insert into libros(titulo,autor,editorial,precio,cantidad)
values('Cervantes y el quijote','Bjyo Casares- J.L. Borges','Paidos',null,100);

insert into libros(titulo,autor,editorial,precio,cantidad)
values('Manual de PHP', 'J.C. Paez', 'Siglo XXI',31.80,120);

insert into libros(titulo,autor,editorial,precio,cantidad)
values('Harry Potter y la piedra filosofal','J.K. Rowling','Emece',45.00,90);

insert into libros(titulo,autor,editorial,precio,cantidad)
values('Harry Potter y la camara secreta','J.K. Rowling','Emece',null,100);

insert into libros(titulo,autor,editorial,precio,cantidad)
values('Alicia en el pais de las maravillas','Lewis Carroll','Paidos',22.50,200);

insert into libros(titulo,autor,editorial,precio,cantidad)
values('PHP de la A a la Z',null,null,null,0);
```

-- Queremos saber la cantidad de libros de cada editorial, utilizando la cláusula "group by":

```
select editorial, count(*)
from libros
group by editorial;
```

-- Obtenemos la cantidad libros con precio no nulo de cada editorial:

```
select editorial, count(precio)
from libros
group by editorial;
```

--Para conocer el total en dinero de los libros agrupados por editorial, tipeamos:

```
select editorial, sum(precio)
from libros
group by editorial;
```

-- Obtenemos el máximo y mínimo valor de los libros agrupados por editorial,

-- en una sola sentencia:

```
select editorial,
```

```
    max(precio) as mayor,
```

```
    min(precio) as menor
```

```
from libros
```

```
group by editorial;
```

-- Calculamos el promedio del valor de los libros agrupados por editorial:

```
select editorial, avg(precio)
```

```
from libros
```

```
group by editorial;
```

-- Es posible limitar la consulta con "where". Vamos a contar y agrupar por editorial

-- considerando solamente los libros cuyo precio es menor a 30 euros:

```
select editorial, count(*)
```

```
from libros
```

```
where precio<30
```

```
group by editorial;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL:11
54   from libros
55   group by editorial;
56
57 -- Calculamos el promedio del valor de los libros agrupados por editorial:
58 select editorial, avg(precio)
59   from libros
60   group by editorial;
61
62 -- Es posible limitar la consulta con "where". Vamos a contar y agrupar por editorial
63 -- considerando solamente los libros cuyo precio es menor a 30 pesos:
64 select editorial, count(*)
65   from libros
66   where precio<30
67   group by editorial;
```

Data Output Explain Messages Notifications Query History

	editorial character varying (15)	count bigint
1	Paidos	1
2	Emece	2
3	Planeta	1

33 - Seleccionar grupos (having)

Así como la cláusula "where" permite seleccionar (o rechazar) registros individuales; la cláusula "having" permite seleccionar (o rechazar) un grupo de registros.

Si queremos saber la cantidad de libros agrupados por editorial usamos la siguiente instrucción ya aprendida:

```
select editorial, count(*)
```

```
from libros
```

```
group by editorial;
```

Si queremos saber la cantidad de libros agrupados por editorial pero considerando sólo algunos grupos, por ejemplo, los que devuelvan un valor mayor a 2, usamos la siguiente instrucción:

```
select editorial, count(*) from libros
```

```
group by editorial
```

```
having count(*)>2;
```

Se utiliza "having", seguido de la condición de búsqueda, para seleccionar ciertas filas retornadas por la cláusula "group by".

Veamos otros ejemplos. Queremos el promedio de los precios de los libros agrupados por editorial, pero solamente de aquellos grupos cuyo promedio supere los 25 euros:

```
select editorial, avg(precio) from libros
```

```
group by editorial
```

```
having avg(precio)>25;
```

En algunos casos es posible confundir las cláusulas "where" y "having". Queremos contar los registros agrupados por editorial sin tener en cuenta a la editorial "Planeta".

Analicemos las siguientes sentencias:

```
select editorial, count(*) from libros
```

```
where editorial<>'Planeta'
```

```
group by editorial;
```

```
select editorial, count(*) from libros
```

```
group by editorial
```

```
having editorial<>'Planeta';
```

Ambas devuelven el mismo resultado, pero son diferentes. La primera, selecciona todos los registros rechazando los de editorial "Planeta" y luego los agrupa para contarlos. La segunda, selecciona todos los registros, los agrupa para contarlos y finalmente rechaza fila con la cuenta correspondiente a la editorial "Planeta".

No debemos confundir la cláusula "where" con la cláusula "having"; la primera establece condiciones para la selección de registros de un "select"; la segunda establece condiciones para la selección de registros de una salida "group by".

Veamos otros ejemplos combinando "where" y "having". Queremos la cantidad de libros, sin considerar los que tienen precio nulo, agrupados por editorial, sin considerar la editorial "Planeta":

```
select editorial, count(*) from libros  
where precio is not null  
group by editorial  
having editorial<>'Planeta';
```

Aquí, selecciona los registros rechazando los que no cumplan con la condición dada en "where", luego los agrupa por "editorial" y finalmente rechaza los grupos que no cumplen con la condición dada en el "having".

Se emplea la cláusula "having" con funciones de agrupamiento, esto no puede hacerlo la cláusula "where". Por ejemplo queremos el promedio de los precios agrupados por editorial, de aquellas editoriales que tienen más de 2 libros:

```
select editorial, avg(precio) from libros  
group by editorial  
having count(*) > 2;
```

Podemos encontrar el mayor valor de los libros agrupados y ordenados por editorial y seleccionar las filas que tengan un valor menor a 100 y mayor a 30:

```
select editorial, max(precio) as mayor  
from libros  
group by editorial  
having min(precio)<100 and  
min(precio)>30  
order by editorial;
```

Entonces, usamos la cláusula "having" para restringir las filas que devuelve una salida "group by". Va siempre después de la cláusula "group by" y antes de la cláusula "order by" si la hubiere.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
    codigo serial,
```

```
    titulo varchar(40),
```

```
    autor varchar(30),
```

```
    editorial varchar(15),
```

```
    precio decimal(5,2),
```

```
    cantidad smallint,
```

```
    primary key(codigo)
```

```
);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('El aleph','Borges','Planeta',35,null);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('Martin Fierro','Jose Hernandez','Emece',22.20,200);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('Martin Fierro','Jose Hernandez','Planeta',40,200);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('Antologia poetica','J.L. Borges','Planeta',null,150);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('Aprenda PHP','Mario Molina','Emece',18,null);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('Manual de PHP', 'J.C. Paez', 'Siglo XXI',56,120);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('Cervantes y el quijote','Btoy Casares- J.L. Borges','Paidos',null,100);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
```

```
values('Harry Potter y la piedra filosofal','J.K. Rowling',default,45.00,90);
```

```
insert into libros(titulo,autor,editorial,precio,cantidad)
values('Harry Potter y la camara secreta','J.K. Rowling','Emece',null,100);

insert into libros(titulo,autor,editorial,precio,cantidad)
values('Alicia en el pais de las maravillas','Lewis Carroll','Paidos',42,80);

insert into libros(titulo,autor,editorial,precio,cantidad)
values('PHP de la A a la Z',null,null,110,0);

insert into libros(titulo,autor,editorial,precio,cantidad)
values('Uno','Richard Bach','Planeta',25,null);
```

-- Queremos saber la cantidad de libros agrupados por editorial pero considerando
-- sólo algunos grupos, por ejemplo, los que devuelvan un valor mayor a 2,
-- usamos la siguiente instrucción:

```
select editorial, count(*) from libros
group by editorial
having count(*)>2;
```

-- Queremos el promedio de los precios de los libros agrupados por editorial,
-- pero solamente de aquellos grupos cuyo promedio supere los 25 euros:

```
select editorial, avg(precio) from libros
group by editorial
having avg(precio)>25;
```

-- Queremos la cantidad de libros, sin considerar los que tienen precio
-- nulo (where), agrupados por editorial (group by),
-- sin considerar la editorial "Planeta" (having):

```
select editorial, count(*) from libros
where precio is not null
group by editorial
having editorial<>'Planeta';
```

-- Necesitamos el promedio de los precios agrupados por editorial,

-- de aquellas editoriales que tienen más de 2 libros:

```
select editorial, avg(precio) from libros
```

```
group by editorial
```

```
having count(*) > 2;
```

-- Buscamos el mayor valor de los libros agrupados y ordenados

--por editorial y seleccionamos las filas que tienen un valor

-- menor a 100 y mayor a 30:

```
select editorial, max(precio) as mayor
```

```
from libros
```

```
group by editorial
```

```
having max(precio)<100 and
```

```
max(precio)>30
```

```
order by editorial;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL terminal window titled "bd1 on postgres@PostgreSQL 11". The query number 65 is highlighted. The terminal displays two SQL queries. The first query selects the average price of books grouped by publisher, filtering for publishers with more than 2 books. The second query selects the maximum price of books grouped by publisher, filtering for prices between 30 and 100. The results are shown in a table with columns "seccion" and "promediodesueldo".

```
64
65 -- Buscamos el mayor valor de los libros agrupados y ordenados
66 --por editorial y seleccionamos las filas que tienen un valor
67 -- menor a 100 y mayor a 30:
68 select editorial, max(precio) as mayor
69   from libros
70   group by editorial
71   having max(precio)<100 and
72     max(precio)>30
73   order by editorial;
74
```

	seccion	promediodesueldo
1	Administracion	3500.00
2	Secretaria	2200.00
3	Sistemas	4000.00
4	Gerencia	5000.00

34 - Registros duplicados (distinct)

Con la cláusula "distinct" se especifica que los registros con ciertos datos duplicados sean obviadas en el resultado. Por ejemplo, queremos conocer todos los autores de los cuales tenemos libros, si utilizamos esta sentencia:

```
select autor from libros;
```

Aparecen repetidos. Para obtener la lista de autores sin repetición usamos:

```
select distinct autor from libros;
```

También podemos tippear:

```
select autor from libros
```

```
group by autor;
```

Note que en los tres casos anteriores aparece "null" como un valor para "autor". Si sólo queremos la lista de autores conocidos, es decir, no queremos incluir "null" en la lista, podemos utilizar la sentencia siguiente:

```
select distinct autor from libros
```

```
where autor is not null;
```

Para contar los distintos autores, sin considerar el valor "null" usamos:

```
select count(distinct autor)
```

```
from libros;
```

Note que si contamos los autores sin "distinct", no incluirá los valores "null" pero si los repetidos:

```
select count(autor)
```

```
from libros;
```

Esta sentencia cuenta los registros que tienen autor.

Podemos combinarla con "where". Por ejemplo, queremos conocer los distintos autores de la editorial "Planeta":

```
select distinct autor from libros
```

```
where editorial='Planeta';
```

También puede utilizarse con "group by" para contar los diferentes autores por editorial:

```
select editorial, count(distinct autor)
```

```
from libros
```

```
group by editorial;
```

La cláusula "distinct" afecta a todos los campos presentados. Para mostrar los títulos y editoriales de los libros sin repetir títulos ni editoriales, usamos:

```
select distinct titulo,editorial
```

```
from libros
```

```
order by titulo;
```

Note que los registros no están duplicados, aparecen títulos iguales pero con editorial diferente, cada registro es diferente.

Entonces, "distinct" elimina registros duplicados.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
codigo serial,
```

```
titulo varchar(40),
```

```
autor varchar(30),
```

```
editorial varchar(15),
```

```
primary key(codigo)
```

```
);
```

```
insert into libros(titulo,autor,editorial)
```

```
values('El aleph','Borges','Planeta');
```

```
insert into libros(titulo,autor,editorial)
```

```
values('Martin Fierro','Jose Hernandez','Emece');
```

```
insert into libros(titulo,autor,editorial)
```

```
values('Martin Fierro','Jose Hernandez','Planeta');
```

```
insert into libros(titulo,autor,editorial)
```

```
values('Antologia poetica','Borges','Planeta');
```

```
insert into libros(titulo,autor,editorial)
```

```
values('Aprenda PHP','Mario Molina','Emece');

insert into libros(titulo,autor,editorial)

values('Aprenda PHP','Lopez','Emece');

insert into libros(titulo,autor,editorial)

values('Manual de PHP', 'J. Paez', null);

insert into libros(titulo,autor,editorial)

values('Cervantes y el quijote',null,'Paidos');

insert into libros(titulo,autor,editorial)

values('Harry Potter y la piedra filosofal','J.K. Rowling','Emece');

insert into libros(titulo,autor,editorial)

values('Harry Potter y la camara secreta','J.K. Rowling','Emece');

insert into libros(titulo,autor,editorial)

values('Alicia en el pais de las maravillas','Lewis Carroll','Paidos');

insert into libros(titulo,autor,editorial)

values('Alicia en el pais de las maravillas','Lewis Carroll','Planeta');

insert into libros(titulo,autor,editorial)

values('PHP de la A a la Z',null,null);

insert into libros(titulo,autor,editorial)

values('Uno','Richard Bach','Planeta');
```

--Para obtener la lista de autores sin repetición tipeamos:

```
select distinct autor from libros;
```

-- Note que aparece "null" como un valor para "autor".

-- Para obtener la lista de autores conocidos, es decir, no incluyendo

-- "null" en la lista:

```
select distinct autor from libros
```

```
where autor is not null;
```

-- Contamos los distintos autores:

```
select count(distinct autor)  
from libros;
```

-- Queremos los nombres de las editoriales sin repetir:

```
select distinct editorial from libros;
```

-- Queremos saber la cantidad de editoriales distintas:

```
select count(distinct editorial) from libros;
```

-- La combinamos con "where" para obtener los distintos

-- autores de la editorial "Planeta":

```
select distinct autor from libros
```

```
where editorial='Planeta';
```

-- Contamos los distintos autores que tiene cada editorial

-- empleando "group by":

```
select editorial,count(distinct autor)
```

```
from libros
```

```
group by editorial;
```

-- Mostramos los títulos y editoriales de los libros sin

-- repetir títulos ni editoriales:

```
select distinct titulo,editorial
```

```
from libros
```

```
order by titulo;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL 11
32   values('Alicia en el país de las maravillas','Lewis Carroll','Paidos');
33   insert into libros(título,autor,editorial)
34   values('Alicia en el país de las maravillas','Lewis Carroll','Planeta');
35   insert into libros(título,autor,editorial)
36   values('PHP de la A a la Z',null,null);
37   insert into libros(título,autor,editorial)
38   values('Uno','Richard Bach','Planeta');
39
40 --Para obtener la lista de autores sin repetición tipeamos:
41 select distinct autor from libros;
42
```

Data Output Explain Messages Notifications Query History

	autor
1	[null]
2	Lewis Carroll
3	Richard Bach
4	Mario Molina
5	J. Paez
6	J.K. Rowling
7	Lopez
8	Borges
9	Jose Hernandez

35 - Clave primaria compuesta

Las claves primarias pueden ser simples, formadas por un solo campo o compuestas, más de un campo.

Recordemos que una clave primaria identifica 1 solo registro en una tabla.

Para un valor del campo clave existe solamente 1 registro. Los valores no se repiten ni pueden ser nulos.

Existe un parking que almacena cada día los datos de los vehículos que ingresan en la tabla llamada "vehiculos" con los siguientes campos:

- patente char(6) not null,
- tipo char (1), 'a'= auto, 'm'=moto,
- horallegada time,
- horasalida time,

Necesitamos definir una clave primaria para una tabla con los datos descriptos arriba. No podemos usar solamente la patente porque un mismo auto puede ingresar más de una vez en el día en el parking; tampoco podemos usar la hora de entrada porque varios autos pueden ingresar a una misma hora.

Tampoco sirven los otros campos.

Como ningún campo, por si sólo cumple con la condición para ser clave, es decir, debe identificar un solo registro, el valor no puede repetirse, debemos usar 2 campos.

Definimos una clave compuesta cuando ningún campo por si solo cumple con la condición para ser clave.

En este ejemplo, un auto puede ingresar varias veces en un día al parking, pero siempre será a distinta hora.

Usamos 2 campos como clave, la patente junto con la hora de llegada, así identificamos únicamente cada registro.

Para establecer más de un campo como clave primaria usamos la siguiente sintaxis:

```
create table vehiculos(  
    patente char(6) not null,  
    tipo char(1)--'a'=auto, 'm'=moto  
    horallegada time,  
    horasalida time,
```

```
primary key(patente,horallegada)
```

```
);
```

Nombramos los campos que formarán parte de la clave separados por comas.

Al ingresar los registros, PostgreSQL controla que los valores para los campos establecidos como clave primaria no estén repetidos en la tabla; si estuviesen repetidos, muestra un mensaje y la inserción no se realiza. Lo mismo sucede si realizamos una actualización.

Entonces, si un solo campo no identifica únicamente un registro podemos definir una clave primaria compuesta, es decir formada por más de un campo.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists vehiculos;
```

```
create table vehiculos(
```

```
    patente char(6) not null,
```

```
    tipo char(1),--'a'=auto, 'm'=moto
```

```
    horallegada time,
```

```
    horasalida time,
```

```
    primary key(patente,horallegada)
```

```
);
```

```
insert into vehiculos values('AIC124','a','8:05','12:30');
```

```
insert into vehiculos values('CAA258','a','8:05',null);
```

```
insert into vehiculos values('DSE367','m','8:30','18:00');
```

```
insert into vehiculos values('FGT458','a','9:00',null);
```

```
insert into vehiculos values('AIC124','a','16:00',null);
```

```
insert into vehiculos values('LOI587','m','18:05','19:55');
```

-- Si intentamos ingresar un registro con clave primaria repetida:

```
insert into vehiculos values('LOI587','m','18:05',null);
```

-- aparece un mensaje de error y la inserción no se realiza.

-- Si ingresamos un registro repitiendo el valor de uno de los

-- campos que forman parte de la clave, si lo acepta:

```
insert into vehiculos values('LOI587','m','21:30',null);
```

-- Si intentamos actualizar un registro repitiendo la clave primaria:

```
update vehiculos set horallegada='8:05'
```

```
where patente='AIC124' and horallegada='16:00';
```

-- aparece un mensaje de error y la actualización no se realiza.

-- Recordemos que los campos que forman parte de la clave primaria

-- no aceptan valores nulos, aunque no se haya aclarado en la definición de la tabla:

```
insert into vehiculos values('HUO690','m',null,null);
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL terminal window titled "bd1 on postgres@PostgreSQL 11". The command history (lines 11-21) includes several INSERT statements into the "vehiculos" table. Line 18 contains a comment: "-- Si intentamos ingresar un registro con clave primaria repetida:". Lines 19 and 20 show an attempt to insert a row with a primary key value ('LOI587') that already exists in the table. A red box highlights this attempt and the resulting error message. The error message is: "ERROR: llave duplicada viola restricción de unicidad «vehiculos_pkey». DETAIL: Ya existe la llave (patente, horallegada)=(LOI587, 18:05:00). SQL state: 23505". Below the terminal window, there is a navigation bar with tabs: Data Output, Explain, Messages, Notifications, and Query History. The "Messages" tab is currently selected.

```
11 insert into vehiculos values('AIC124','a','8:05','12:30');
12 insert into vehiculos values('CAA258','a','8:05',null);
13 insert into vehiculos values('DSE367','m','8:30','18:00');
14 insert into vehiculos values('FGT458','a','9:00',null);
15 insert into vehiculos values('AIC124','a','16:00',null);
16 insert into vehiculos values('LOI587','m','18:05','19:55');
17
18 -- Si intentamos ingresar un registro con clave primaria repetida:
19 insert into vehiculos values('LOI587','m','18:05',null);
20 -- aparece un mensaje de error y la inserción no se realiza.
21
```

Data Output Explain Messages Notifications Query History

ERROR: llave duplicada viola restricción de unicidad «vehiculos_pkey»
DETAIL: Ya existe la llave (patente, horallegada)=(LOI587, 18:05:00).
SQL state: 23505

36 - Restricción check

La restricción "check" especifica los valores que acepta un campo, evitando que se ingresen valores inapropiados.

La sintaxis básica es la siguiente:

```
alter table NOMBRETABLA  
add constraint NOMBRECONSTRAINT  
check CONDICION;
```

Trabajamos con la tabla "libros" de una librería que tiene los siguientes campos: codigo, titulo, autor, editorial, preciomin (que indica el precio para los minoristas) y preciomay (que indica el precio para los mayoristas).

Los campos correspondientes a los precios (minorista y mayorista) se definen de tipo decimal(5,2), es decir, aceptan valores entre -999.99 y 999.99. Podemos controlar que no se ingresen valores negativos para dichos campos agregando una restricción "check":

```
alter table libros  
add constraint CK_libros_precio_positivo  
check (preciomin>=0 and preciomay>=0);
```

Este tipo de restricción verifica los datos cada vez que se ejecuta una sentencia "insert" o "update", es decir, actúa en inserciones y actualizaciones.

Si la tabla contiene registros que no cumplen con la restricción que se va a establecer, la restricción no se puede establecer, hasta que todos los registros cumplan con dicha restricción.

La condición puede hacer referencia a otros campos de la misma tabla. Por ejemplo, podemos controlar que el precio mayorista no sea mayor al precio minorista:

```
alter table libros  
add constraint CK_libros_precioiminmay  
check (preciomay<=preciomin);
```

Por convención, cuando demos el nombre a las restricciones "check" seguiremos la misma estructura: comenzamos con "CK", seguido del nombre de la tabla, del campo y alguna palabra con la cual podamos identificar fácilmente de qué se trata la restricción, por si tenemos varias restricciones "check" para el mismo campo.

Un campo puede tener varias restricciones "check" y una restricción "check" puede incluir varios campos.

Si un campo permite valores nulos, "null" es un valor aceptado aunque no esté incluido en la condición de restricción.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
    codigo serial,
```

```
    titulo varchar(40),
```

```
    autor varchar(30),
```

```
    editorial varchar(15),
```

```
    preciomin decimal(5,2),
```

```
    preciomay decimal(5,2),
```

```
    primary key(codigo)
```

```
);
```

```
insert into libros (titulo,autor,editorial,preciomin,preciomay)
```

```
values ('Aprenda PHP','Mario Molina','Siglo XXI', 48, 53);
```

-- Agregamos una restricción "check" para asegurar que los valores de los

-- campos correspondientes a precios no puedan ser negativos:

```
alter table libros
```

```
add constraint CK_libros_precios_positivo
```

```
check (preciomin>=0 and preciomay>=0);
```

-- Si intentamos ingresar un valor inválido para algún campo correspondiente

-- al precio, que vaya en contra de la restricción, por ejemplo el valor "-15"

-- aparecerá un mensaje de error indicando que hay conflicto con la restricción

-- creada anteriormente y la inserción no se realiza.

```
insert into libros (titulo,autor,editorial,preciomin,preciomay)
```

```
values ('Python para todos','Rodriguez','Siglo XXI', -10, 40);
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL:11
 3  create table libros(
 4    codigo serial,
 5    titulo varchar(40),
 6    autor varchar(30),
 7    editorial varchar(15),
 8    preciomin decimal(5,2),
 9    preciomay decimal(5,2),
10    primary key(codigo)
11  );
12
13  insert into libros (titulo,autor,editorial,preciomin,preciomay)
14    values ('Aprenda PHP','Mario Molina','Siglo XXI', 48, 53);
15
16  -- Agregamos una restricción "check" para asegurar que los valores de los
17  -- campos correspondientes a precios no puedan ser negativos:
18  alter table libros
19    add constraint CK_libros_precios_positivo
20    check (preciomin>=0 and preciomay>=0);
21
22  -- Si intentamos ingresar un valor inválido para algún campo correspondiente
23  -- al precio, que vaya en contra de la restricción, por ejemplo el valor "-15"
24  -- aparecerá un mensaje de error indicando que hay conflicto con la restricción
25  -- creada anteriormente y la inserción no se realiza.
26  insert into libros (titulo,autor,editorial,preciomin,preciomay)
27    values ('Python para todos','Rodriguez','Siglo XXI', -10, 40);
28
```

Data Output Explain Messages Notifications Query History

```
ERROR: el nuevo registro para la relación «libros» viola la restricción «check» «ck_libros_precios_positivo»
DETAIL: La fila que falla contiene (2, Python para todos, Rodriguez, Siglo XXI, -10.00, 40.00).
SQL state: 23514
```

37 - Restricción primary key

Hemos visto la restricción que se aplica a los campos con "check".

Ahora veremos las restricciones que se aplican a las tablas, que aseguran valores únicos para cada registro.

Hay 2 tipos: 1) primary key y 2) unique.

Anteriormente, para establecer una clave primaria para una tabla empleábamos la siguiente sintaxis al crear la tabla, por ejemplo:

```
create table libros(
```

```
    codigo int not null,
```

```
    titulo varchar(30),
```

```
    autor varchar(30),
```

```
    editorial varchar(20),
```

```
    primary key(codigo)
```

```
);
```

Cada vez que establecíamos la clave primaria para la tabla, PostgreSQL creaba automáticamente una restricción "primary key" para dicha tabla. Dicha restricción, a la cual no le dábamos un nombre, recibía un nombre dado por PostgreSQL, por ejemplo 'libros_pkey'

Podemos agregar una restricción "primary key" a una tabla existente con la sintaxis básica siguiente:

```
alter table NOMBRERESTRICA
```

```
add constraint NOMBRECONSTRAINT
```

```
primary key (CAMPO,...);
```

En el siguiente ejemplo definimos una restricción "primary key" para nuestra tabla "libros" para asegurarnos que cada libro tendrá un código diferente y único:

```
alter table libros
```

```
add constraint PK_libros_codigo
```

```
primary key(codigo);
```

Con esta restricción, si intentamos ingresar un registro con un valor para el campo "codigo" que ya existe o el valor "null", aparece un mensaje de error, porque no se permiten valores duplicados ni nulos. Igualmente, si actualizamos.

Por convención, cuando demos el nombre a las restricciones "primary key" seguiremos el formato "PK_NOMBRETABLA_NOMBRECAMPO".

Sabemos que cuando agregamos una restricción a una tabla que contiene información, PostgreSQL controla los datos existentes para confirmar que cumplen las exigencias de la restricción, si no los cumple, la restricción no se aplica y aparece un mensaje de error. Por ejemplo, si intentamos definir la restricción "primary key" para "libros" y hay registros con códigos repetidos o con un valor "null", la restricción no se establece.

PostgreSQL permite definir solamente una restricción "primary key" por tabla, que asegura la unicidad de cada registro de una tabla.

Si ejecutamos:

```
select *  
from information_schema.table_constraints  
where table_name = 'libros';
```

podemos ver las restricciones "primary key" (y todos los tipos de restricciones) de dicha tabla.

Un campo con una restricción "primary key" puede tener una restricción "check".

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
codigo int not null,
```

```
titulo varchar(40),
```

```
autor varchar(30),
```

```
editorial varchar(15),
```

```
primary key (codigo)
```

```
);
```

-- Veamos la restricción "primary key" que creó automáticamente PosgreSQL:

```
select *  
from information_schema.table_constraints  
where table_name = 'libros';
```

-- Vamos a eliminar la tabla y la crearemos nuevamente, sin establecer la clave primaria:

```
drop table libros;
```

```
create table libros(
```

```
    codigo int not null,
```

```
    titulo varchar(40),
```

```
    autor varchar(30),
```

```
    editorial varchar(15)
```

```
);
```

-- Definimos una restricción "primary key" para nuestra tabla "libros" para asegurarnos

-- que cada libro tendrá un código diferente y único:

```
alter table libros
```

```
    add constraint PK_libros_codigo
```

```
    primary key(codigo);
```

-- Veamos la información respecto a ella:

```
select *
```

```
from information_schema.table_constraints
```

```
where table_name = 'libros';
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bdt on postgres@PostgreSQL:11
26 -- Definimos una restricción "primary key" para nuestra tabla "libros" para asegurarnos
27 -- que cada libro tendrá un código diferente y único:
28 alter table libros
29     add constraint PK_libros_codigo
30     primary key(codigo);
31
32 -- Veamos la información respecto a ella:
33 select *
34 from information_schema.table_constraints
35 where table_name = 'libros';
36
```

	constraint_catalog	constraint_schema	constraint_name	table_catalog	table_schema	table_name	constraint_type
1	bdt	public	libros_pkey	bdt	public	libros	PRIMARY KEY
2	bdt	public	2200_25738_1_not_null	bdt	public	libros	CHECK

38 - Restricción unique

Anteriormente aprendimos la restricción "primary key", otra restricción para las tablas es "unique".

La restricción "unique" impide la duplicación de claves alternas (no primarias), es decir, especifica que dos registros no puedan tener el mismo valor en un campo. Se permiten valores nulos. Se pueden aplicar varias restricciones de este tipo a una misma tabla, y pueden aplicarse a uno o varios campos que no sean clave primaria.

Se emplea cuando ya se estableció una clave primaria (como un número de legajo) pero se necesita asegurar que otros datos también sean únicos y no se repitan (como número de documento).

La sintaxis general es la siguiente:

```
alter table NOMBRERESTRICCION  
add constraint NOMBRERESTRICCION  
unique (CAMPO);
```

Ejemplo:

```
alter table alumnos  
add constraint UQ_alumnos_documento  
unique (documento);
```

En el ejemplo anterior se agrega una restricción "unique" sobre el campo "documento" de la tabla "alumnos", esto asegura que no se pueda ingresar un documento si ya existe. Esta restricción permite valores nulos, así que si se ingresa el valor "null" para el campo "documento", se acepta.

Por convención, cuando demos el nombre a las restricciones "unique" seguiremos la misma estructura: "UQ_NOMBRETABLA_NOMBRECAMPO". Quizá parezca innecesario colocar el nombre de la tabla, pero cuando empleemos varias tablas verá que es útil identificar las restricciones por tipo, tabla y campo.

Recuerde que cuando agregamos una restricción a una tabla que contiene información, PostgreSQL controla los datos existentes para confirmar que cumplen la condición de la restricción, si no los cumple, la restricción no se aplica y aparece un mensaje de error. En el caso del ejemplo anterior, si la tabla contiene números de documento duplicados, la restricción no podrá establecerse; si podrá establecerse si tiene valores nulos.

PostgreSQL controla la entrada de datos en inserciones y actualizaciones evitando que se ingresen valores duplicados.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists alumnos;
```

```
create table alumnos(
    legajo char(4) not null,
    apellido varchar(20),
    nombre varchar(20),
    documento char(8)
);
```

-- Agregamos una restricción "primary" para el campo "legajo":

```
alter table alumnos
add constraint PK_alumnos_legajo
primary key(legajo);
```

-- Agregamos una restricción "unique" para el campo "documento":

```
alter table alumnos
add constraint UQ_alumnos_documento
unique (documento);
```

-- Ingresamos algunos registros:

```
insert into alumnos values('A111','Lopez','Ana','22222222');
```

```
insert into alumnos values('A123','Garcia','Maria','23333333');
```

-- Si intentamos ingresar un legajo o documento repetido, aparece un mensaje de error.

-- Veamos las restricciones:

```
select *
from information_schema.table_constraints
where table_name = 'alumnos';
```

-- Aparecen las dos restricciones creadas anteriormente.

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL 11
19
20 -- Ingresamos algunos registros:
21 insert into alumnos values('A111','Lopez','Ana','22222222');
22 insert into alumnos values('A123','Garcia','Maria','23333333');
23 -- Si intentamos ingresar un legajo o documento repetido, aparece un mensaje de error.
24
25 -- Veamos las restricciones:
26 select *
27   from information_schema.table_constraints
28  where table_name = 'alumnos';
29 -- Aparecen las dos restricciones creadas anteriormente.
```

	Data Output	Explain	Messages	Notifications	Query History		
1	constraint_catalog character varying	constraint_schema character varying	constraint_name character varying	table_catalog character varying	table_schema character varying	table_name character varying	constraint_type character varying
2	bd1	public	pk_alumnos_legajo	bd1	public	alumnos	PRIMARY KEY
3	bd1	public	uq_alumnos_documento	bd1	public	alumnos	UNIQUE
4	bd1	public	2200_25743_1_not_null	bd1	public	alumnos	CHECK

39 - Eliminar restricciones (alter table - drop constraint)

Para eliminar una restricción, la sintaxis básica es la siguiente:

```
alter table NOMBRETABLA  
drop constraint NOMBRERESTRICCION;
```

Para eliminar la restricción "CK_libros_precio_positivo" de la tabla libros tipeamos:

```
alter table libros  
drop constraint CK_libros_precio_positivo;
```

Cuando eliminamos una tabla, todas las restricciones que fueron establecidas en ella, se eliminan también.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
codigo int not null,
```

```
titulo varchar(40),
```

```
autor varchar(30),
```

```
editorial varchar(15),
```

```
precio decimal(6,2)
```

```
);
```

```
-- Definimos una restricción "primary key" para nuestra tabla "libros"
```

```
--para asegurarnos que cada libro tendrá un código diferente y único:
```

```
alter table libros  
add constraint PK_libros_codigo  
primary key(codigo);
```

```
-- Definimos una restricción "check" para asegurarnos que el precio
```

```
-- no será negativo:
```

```
alter table libros
```

```
add constraint CK_libros_precio
```

```
check (precio>=0);
```

```
-- Vemos las restricciones:
```

```
select *
```

```
from information_schema.table_constraints
```

```
where table_name = 'libros';
```

```
-- Aparecen 2 restricciones, 1 "check" y 1 "primary key".
```

```
-- Eliminamos la restricción "PK_libros_codigo":
```

```
alter table libros
```

```
drop constraint PK_libros_codigo;
```

```
-- Vemos si se eliminó:
```

```
select *
```

```
from information_schema.table_constraints
```

```
where table_name = 'libros';
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL:11
27  -- Aparecen 2 restricciones, 1 "check" y 1 "primary key".
28
29  -- Eliminamos la restricción "PK_libros_codigo":
30  alter table libros
31    drop constraint PK_libros_codigo;
32
33  -- Vemos si se eliminaron:
34  select *
35    from information_schema.table_constraints
36  where table_name = 'libros';
37
```

Data Output						
	constraint_catalog character varying	constraint_schema character varying	constraint_name character varying	table_catalog character varying	table_schema character varying	constraint_type character varying
1	bd1	public	ck_libros_precio	bd1	public	libros
2	bd1	public	2200_25762_l_not_n...	bd1	public	libros

40 - Índice de una tabla.

Para facilitar la obtención de información de una tabla se utilizan índices.

El índice de una tabla desempeña la misma función que el índice de un libro: permite encontrar datos rápidamente; en el caso de las tablas, localiza registros.

Una tabla se indexa por un campo (o varios).

El índice es un tipo de archivo con 2 entradas: un dato (un valor de algún campo de la tabla) y un puntero.

Un índice posibilita el acceso directo y rápido haciendo más eficiente las búsquedas. Sin índice, se debe recorrer secuencialmente toda la tabla para encontrar un registro.

El objetivo de un índice es acelerar la recuperación de información.

La desventaja es que consume espacio en el disco y las inserciones y borrados de registros son más lentas.

La indexación es una técnica que optimiza el acceso a los datos, mejora el rendimiento acelerando las consultas y otras operaciones. Es útil cuando la tabla contiene miles de registros.

Los índices se usan para varias operaciones:

- para buscar registros rápidamente.
- para recuperar registros de otras tablas empleando "join".

Es importante identificar el o los campos por los que sería útil crear un índice, aquellos campos por los cuales se realizan operaciones de búsqueda con frecuencia.

Hay distintos tipos de índices, a saber:

- 1) "primary key": es el que definimos como clave primaria. Los valores indexados deben ser únicos y además no pueden ser nulos. PostgreSQL le da el nombre "PRIMARY". Una tabla solamente puede tener una clave primaria.
- 2) "index": crea un índice común, los valores no necesariamente son únicos y aceptan valores "null". Podemos darle un nombre, si no se lo damos, se coloca uno por defecto. "key" es sinónimo de "index". Puede haber varios por tabla.
- 3) "unique": crea un índice para los cuales los valores deben ser únicos y diferentes, aparece un mensaje de error si intentamos agregar un registro con un valor ya existente. Permite valores nulos y pueden definirse varios por tabla. Podemos darle un nombre, si no se lo damos, se coloca uno por defecto.

Todos los índices pueden ser multicolumna, es decir, pueden estar formados por más de 1 campo.

En las siguientes lecciones aprenderemos sobre cada uno de ellos.

Los nombres de índices aceptan todos los caracteres.

Una tabla puede ser indexada por campos de tipo numérico o de tipo carácter. También se puede indexar por un campo que contenga valores NULL, excepto los PRIMARY.

41 - Tipos de índices (create y drop)

Dijimos que hay 3 tipos de índices:

El índice llamado primary se crea automáticamente cuando establecemos un campo como clave primaria.

Los valores indexados deben ser únicos y además no pueden ser nulos. Una tabla solamente puede tener una clave primaria. Puede ser multicolumna, es decir, pueden estar formados por más de un campo.

Vamos a otro tipo de índice común. Un índice común se crea con "create index", los valores no necesariamente son únicos y aceptan valores "null". Puede haber varios por tabla.

Vamos a trabajar con nuestra tabla "libros".

```
create table libros(
```

```
    codigo int not null,
```

```
    titulo varchar(40),
```

```
    autor varchar(30),
```

```
    editorial varchar(15),
```

```
    precio decimal(6,2)
```

```
);
```

Un campo por el cual realizamos consultas frecuentemente es "editorial", indexar la tabla por ese campo sería útil.

Creamos un índice:

```
create index I_libros_editorial on libros(editorial);
```

Debemos definir un nombre para el índice (en este caso utilizamos como nomenclatura el carácter I, luego el nombre de la tabla y finalmente el o los nombres del campo por el cual creamos el índice. Luego de la palabra clave on indicamos el nombre de la tabla y entre paréntesis el nombre del campo o los campos por el cual se indexa.

Veamos otro tipo de índice llamado "único". Un índice único se crea con "create unique index", los valores deben ser únicos y diferentes, aparece un mensaje de error si intentamos agregar un registro con un valor ya existente. Permite valores nulos y pueden definirse varios por tabla.

Crearemos un índice único por los campos titulo y editorial:

```
create unique index I_libros_tituloeditorial on libros(titulo,editorial);
```

Para eliminar un índice usamos "drop index". Ejemplo:

```
drop index I_libros_editorial;  
drop index I_libros_tituloeditorial;
```

Se elimina un índice con "drop index" seguido de su nombre.

Podemos eliminar los índices creados, pero no el creado automáticamente con la clave primaria.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;  
  
create table libros(  
  
    codigo serial,  
  
    titulo varchar(40) not null,  
  
    autor varchar(30),  
  
    editorial varchar(15),  
  
    primary key(codigo)  
  
);
```

-- Creamos un índice común por el campo editorial:

```
create index I_libros_editorial on libros(editorial);
```

-- Ahora crearemos un índice único por los campos titulo y editorial:

```
create unique index I_libros_tituloeditorial on libros(titulo,editorial);
```

-- Borramos los dos índices que acabamos de crear:

```
drop index I_libros_editorial;  
drop index I_libros_tituloeditorial;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL 11
1 drop table if exists libros;
2
3 create table libros(
4     codigo serial,
5     titulo varchar(40) not null,
6     autor varchar(30),
7     editorial varchar(15),
8     primary key(codigo)
9 );
10
11 -- Creamos un índice común por el campo editorial:
12 create index I_libros_editorial on libros(editorial);
13
14 -- Ahora crearemos un índice único por los campos título y editorial:
15 create unique index I_libros_tituloeditorial on libros(titulo,editorial);
16
17 -- Borramos los dos índices que acabamos de crear:
18 drop index I_libros_editorial;
19 drop index I_libros_tituloeditorial;
20
```

Data Output Explain Messages Notifications Query History

DROP INDEX

Query returned successfully in 296 msec.

42 - Cláusulas limit y offset del comando select

Las cláusulas "limit" y "offset" se usan para restringir los registros que se retornan en una consulta "select".

La cláusula limit recibe un argumento numérico positivo que indica el número máximo de registros a retornar; la cláusula offset indica el número del primer registro a retornar. El número de registro inicial es 0 (no 1).

Si el limit supera la cantidad de registros de la tabla, se limita hasta el último registro.

Ejemplo:

```
select * from libros limit 4 offset 0;
```

Muestra los primeros 4 registros, 0,1,2 y 3.

Si tipeamos:

```
select * from libros limit 4 offset 5;
```

recuperamos 4 registros, desde el 5 al 8.

Si se coloca solo la cláusula limit retorna tantos registros como el valor indicado, comenzando desde 0. Ejemplo:

```
select * from libros limit 3
```

Muestra los primeros 3 registros.

Es conveniente utilizar la cláusula order by cuando utilizamos limit y offset, por ejemplo:

```
select * from libros order by codigo limit 3;
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
    codigo serial,
```

```
    titulo varchar(50) not null,
```

```
    autor varchar(30),
```

```
    editorial varchar(15),
```

```
    precio decimal(5,2),
```

```
    primary key (codigo)
```

```
);

insert into libros (titulo,autor,editorial,precio)
values('El aleph','Borges','Planeta',15);

insert into libros (titulo,autor,editorial,precio)
values('Martin Fierro','Jose Hernandez','Emece',22.20);

insert into libros (titulo,autor,editorial,precio)
values('Antologia poetica','Borges','Planeta',40);

insert into libros (titulo,autor,editorial,precio)
values('Aprenda PHP','Mario Molina','Emece',18.20);

insert into libros (titulo,autor,editorial,precio)
values('Cervantes y el quijote','Borges','Paidos',36.40);

insert into libros (titulo,autor,editorial,precio)
values('Manual de PHP', 'J.C. Paez', 'Paidos',30.80);

insert into libros (titulo,autor,editorial,precio)
values('Harry Potter y la piedra filosofal','J.K. Rowling','Paidos',45.00);

insert into libros (titulo,autor,editorial,precio)
values('Harry Potter y la camara secreta','J.K. Rowling','Paidos',46.00);

insert into libros (titulo,autor,editorial,precio)
values('Alicia en el pais de las maravillas','Lewis Carroll','Paidos',null);
```

-- Para recuperar 4 libros desde el registro cero tipeamos:

```
select * from libros limit 4 offset 0;
```

-- Para recuperar 4 libros a partir del registro 5:

```
select * from libros limit 4 offset 5;
```

-- Si colocamos solo el limit, éste indica el máximo número de registros

-- a retornar, comenzando desde 0:

```
select * from libros limit 8;
```

-- Mostrar los tres primeros registro ordenados por código:

```
select * from libros order by codigo limit 3;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL-11
34  -- Para recuperar 4 libros a partir del registro 5:
35  select * from libros limit 4 offset 5;
36
37  -- Si colocamos solo el limit, éste indica el máximo número de registros
38  -- a retornar, comenzando desde 0:
39  select * from libros limit 8;
40
41  -- Mostrar los tres primeros registro ordenados por código:
42  select * from libros order by codigo limit 3;
```

Data Output	Explain	Messages	Notifications	Query History
codigo integer	título character varying (50)	autor character varying (30)	editorial character varying (15)	precio numeric (5,2)
1	1 El aleph	Borges	Planeta	15,00
2	2 Martín Fierro	Jose Hernandez	Emece	22,20
3	3 Antología poética	Borges	Planeta	40,00

43 - Trabajar con varias tablas

Hasta el momento hemos trabajado con una sola tabla, pero generalmente, se trabaja con más de una.

Para evitar la repetición de datos y ocupar menos espacio, se separa la información en varias tablas. Cada tabla almacena parte de la información que necesitamos registrar.

Por ejemplo, los datos de nuestra tabla "libros" podrían separarse en 2 tablas, una llamada "libros" y otra "editoriales" que guardará la información de las editoriales.

En nuestra tabla "libros" haremos referencia a la editorial colocando un código que la identifique.

Veamos:

```
create table libros(  
    codigo serial,  
    titulo varchar(40) not null,  
    autor varchar(30) not null default 'Desconocido',  
    codigoeditorial smallint not null,  
    precio decimal(5,2),  
    primary key (codigo)  
);
```

```
create table editoriales(  
    codigo serial,  
    nombre varchar(20) not null,  
    primary key(codigo)  
);
```

De esta manera, evitamos almacenar tantas veces los nombres de las editoriales en la tabla "libros" y guardamos el nombre en la tabla "editoriales"; para indicar la editorial de cada libro agregamos un campo que hace referencia al código de la editorial en la tabla "libros" y en "editoriales".

Al recuperar los datos de los libros con la siguiente instrucción:

```
select * from libros;
```

vemos que en el campo "editorial" aparece el código, pero no sabemos el nombre de la editorial. Para obtener los datos de cada libro, incluyendo el nombre de la editorial, necesitamos consultar ambas tablas, traer información de las dos.

Cuando obtenemos información de más de una tabla decimos que hacemos un "join" (combinación).

Veamos un ejemplo:

```
select * from libros  
join editoriales  
on libros.codigoeditorial=editoriales.codigo;
```

Resumiendo: si distribuimos la información en varias tablas evitamos la redundancia de datos y ocupamos menos espacio físico en el disco. Un join es una operación que relaciona dos o más tablas para obtener un resultado que incluya datos (campos y registros) de ambas; las tablas participantes se combinan según los campos comunes a ambas tablas.

Hay tres tipos de combinaciones. En los siguientes capítulos explicamos cada una de ellas.

44 - Combinación interna (inner join)

Un join es una operación que relaciona dos o más tablas para obtener un resultado que incluya datos (campos y registros) de ambas; las tablas participantes se combinan según los campos comunes a ambas tablas.

Hay tres tipos de combinaciones:

combinaciones internas (inner join o join),
combinaciones externas y
combinaciones cruzadas.

También es posible emplear varias combinaciones en una consulta "select", incluso puede combinarse una tabla consigo misma.

La combinación interna emplea "join", que es la forma abreviada de "inner join". Se emplea para obtener información de dos tablas y combinar dicha información en una salida.

La sintaxis básica es la siguiente:

```
select CAMPOS  
from TABLA1  
join TABLA2  
on CONDICIONdeCOMBINACION;
```

Ejemplo:

```
select * from libros  
join editoriales  
on codigoeditorial=editoriales.codigo;
```

Analicemos la consulta anterior.

- especificamos los campos que aparecerán en el resultado en la lista de selección;
- indicamos el nombre de la tabla luego del "from" ("libros");
- combinamos esa tabla con "join" y el nombre de la otra tabla ("editoriales"); se especifica qué tablas se van a combinar y cómo;
- cuando se combina información de varias tablas, es necesario especificar qué registro de una tabla se combinará con qué registro de la otra tabla, con "on". Se debe especificar la condición para enlazarlas, es decir, el campo por el cual se combinarán, que tienen en común.
"on" hace coincidir registros de ambas tablas basándose en el valor de tal campo, en el ejemplo,

el campo "codigoeditorial" de "libros" y el campo "codigo" de "editoriales" son los que enlazarán ambas tablas. Se emplean campos comunes, que deben tener tipos de datos iguales o similares.

La condición de combinación, es decir, el o los campos por los que se van a combinar (parte "on"), se especifica según las claves primarias y externas.

Note que en la consulta, al nombrar el campo usamos el nombre de la tabla también. Cuando las tablas referenciadas tienen campos con igual nombre, esto es necesario para evitar confusiones y ambigüedades al momento de referenciar un campo. En el ejemplo, si no especificamos "editoriales.codigo" y solamente tipeamos "codigo", PosgreSQL no sabrá si nos referimos al campo "codigo" de "libros" o de "editoriales" y mostrará un mensaje de error indicando que "codigo" es ambiguo.

Entonces, si las tablas que combinamos tienen nombres de campos iguales, DEBE especificarse a qué tabla pertenece anteponiendo el nombre de la tabla al nombre del campo, separado por un punto (.).

Si una de las tablas tiene clave primaria compuesta, al combinarla con la otra, en la cláusula "on" se debe hacer referencia a la clave completa, es decir, la condición referenciará a todos los campos clave que identifican al registro.

Se puede incluir en la consulta join la cláusula "where" para restringir los registros que retorna el resultado; también "order by", etc..

Se emplea este tipo de combinación para encontrar registros de la primera tabla que se correspondan con los registros de la otra, es decir, que cumplan la condición del "on". Si un valor de la primera tabla no se encuentra en la segunda tabla, el registro no aparece.

Para simplificar la sentencia podemos usar un alias para cada tabla:

```
select l.codigo,titulo,autor,nombre  
from libros as l  
join editoriales as e  
on l.codigoeditorial=e.codigo;
```

En algunos casos (como en este ejemplo) el uso de alias es para fines de simplificación y hace más legible la consulta si es larga y compleja, pero en algunas consultas es absolutamente necesario.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(  
    codigo serial,  
    titulo varchar(40),
```

```
autor varchar(30) default 'Desconocido',
codigoeditorial smallint not null,
precio decimal(5,2),
primary key(codigo)
);
```

```
drop table if exists editoriales;
```

```
create table editoriales(
codigo serial,
nombre varchar(20),
primary key (codigo)
);
```

```
insert into editoriales(nombre) values('Planeta');
insert into editoriales(nombre) values('Emece');
insert into editoriales(nombre) values('Siglo XXI');
```

```
insert into libros(titulo,autor,codigoeditorial,precio)
values('El aleph','Borges',2,20);
insert into libros(titulo,autor,codigoeditorial,precio)
values('Martin Fierro','Jose Hernandez',1,30);
insert into libros(titulo,autor,codigoeditorial,precio)
values('Aprenda PHP','Mario Molina',3,50);
insert into libros(titulo,autor,codigoeditorial,precio)
values('Java en 10 minutos',default,3,45);
```

```
-- Recuperamos los datos de libros:
```

```
select* from libros;
```

-- Vemos que en el campo "editorial" aparece el código,
-- pero no sabemos el nombre de la editorial.

-- Realizamos un join para obtener datos de ambas tablas

-- (titulo, autor y nombre de la editorial):

```
select titulo, autor, nombre  
from libros  
join editoriales  
on codigoeditorial=editoriales.codigo;
```

-- Mostramos el código del libro, título, autor, nombre

-- de la editorial y el precio realizando un join y empleando alias:

```
select l.codigo,titulo,autor,nombre,precio  
from libros as l  
join editoriales as e  
on codigoeditorial=e.codigo;
```

-- Realizamos la misma consulta anterior agregando un "where"

-- para obtener solamente los libros de la editorial "Siglo XXI":

```
select l.codigo,titulo,autor,nombre,precio  
from libros as l  
join editoriales as e  
on codigoeditorial=e.codigo  
where e.nombre='Siglo XXI';
```

-- Obtenemos título, autor y nombre de la editorial, esta vez ordenados por título:

```
select titulo,autor,nombre
```

```
from libros as l  
join editoriales as e  
on codigoeditorial=e.codigo  
order by titulo;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL terminal window titled "bd1 on postgres@PostgreSQL 11". The query entered is:

```
-- Obtenemos título, autor y nombre de la editorial, esta vez ordenados por título:  
select título,autor,nombre  
from libros as l  
join editoriales as e  
on codigoeditorial=e.codigo  
order by título;
```

The results are displayed in a table:

	título character varying (40)	autor character varying (30)	nombre character varying (20)
1	Aprenda PHP	Mario Molina	Siglo XXI
2	El eleph	Borges	Emece
3	Java en 10 minutos	Desconocido	Siglo XXI
4	Martín Fierro	José Hernández	Planeta

45 - Combinación externa izquierda (left join)

Vimos que una combinación interna (join) encuentra registros de la primera tabla que se correspondan con los registros de la segunda, es decir, que cumplan la condición del "on" y si un valor de la primera tabla no se encuentra en la segunda tabla, el registro no aparece.

Si queremos saber qué registros de una tabla NO encuentran correspondencia en la otra, es decir, no existe valor coincidente en la segunda, necesitamos otro tipo de combinación, "outer join" (combinación externa).

Las combinaciones externas combinan registros de dos tablas que cumplen la condición, más los registros de la segunda tabla que no la cumplen; es decir, muestran todos los registros de las tablas relacionadas, aún cuando no haya valores coincidentes entre ellas.

Este tipo de combinación se emplea cuando se necesita una lista completa de los datos de una de las tablas y la información que cumple con la condición. Las combinaciones externas se realizan solamente entre 2 tablas.

Hay tres tipos de combinaciones externas: "left outer join", "right outer join" y "full outer join"; se pueden abreviar con "left join", "right join" y "full join" respectivamente.

Vamos a estudiar las primeras.

Se emplea una combinación externa izquierda para mostrar todos los registros de la tabla de la izquierda. Si no encuentra coincidencia con la tabla de la derecha, el registro muestra los campos de la segunda tabla seteados a "null".

En el siguiente ejemplo solicitamos el título y nombre de la editorial de los libros:

```
select titulo,nombre  
from editoriales as e  
left join libros as l  
on codigoeditorial = e.codigo;
```

El resultado mostrará el título y nombre de la editorial; las editoriales de las cuales no hay libros, es decir, cuyo código de editorial no está presente en "libros" aparece en el resultado, pero con el valor "null" en el campo "titulo".

Es importante la posición en que se colocan las tablas en un "left join", la tabla de la izquierda es la que se usa para localizar registros en la tabla de la derecha.

Entonces, un "left join" se usa para hacer coincidir registros en una tabla (izquierda) con otra tabla (derecha); si un valor de la tabla de la izquierda no encuentra coincidencia en la tabla de la derecha, se genera una fila extra (una por cada valor no encontrado) con todos los campos correspondientes a la tabla derecha seteados a "null". La sintaxis básica es la siguiente:

```
select CAMPOS
```

```
from TABLAIZQUIERDA
```

```
left join TABLADERECHA
```

```
on CONDICION;
```

En el siguiente ejemplo solicitamos el título y el nombre la editorial, la sentencia es similar a la anterior, la diferencia está en el orden de las tablas:

```
select titulo,nombre
```

```
from libros as l
```

```
left join editoriales as e
```

```
on codigoeditorial = e.codigo;
```

El resultado mostrará el título del libro y el nombre de la editorial; los títulos cuyo código de editorial no está presente en "editoriales" aparecen en el resultado, pero con el valor "null" en el campo "nombre".

Un "left join" puede tener clausula "where" que restrinja el resultado de la consulta considerando solamente los registros que encuentran coincidencia en la tabla de la derecha, es decir, cuyo valor de código está presente en "libros":

```
select titulo,nombre
```

```
from editoriales as e
```

```
left join libros as l
```

```
on e.codigo=codigoeditorial
```

```
where codigoeditorial is not null;
```

También podemos mostrar las editoriales que NO están presentes en "libros", es decir, que NO encuentran coincidencia en la tabla de la derecha:

```
select titulo,nombre
```

```
from editoriales as e
```

```
left join libros as l
```

```
on e.codigo=codigoeditorial
```

```
where codigoeditorial is null;
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
drop table if exists editoriales;
```

```

create table libros(
    codigo serial,
    titulo varchar(40),
    autor varchar(30) default 'Desconocido',
    codigoeditorial smallint not null,
    precio decimal(5,2),
    primary key(codigo)
);

create table editoriales(
    codigo serial,
    nombre varchar(20),
    primary key (codigo)
);

insert into editoriales(nombre) values('Planeta');

insert into editoriales(nombre) values('Emece');

insert into editoriales(nombre) values('Siglo XXI');

insert into libros(titulo,autor,codigoeditorial,precio)
values('El aleph','Borges',1,20);

insert into libros(titulo,autor,codigoeditorial,precio)
values('Martin Fierro','Jose Hernandez',1,30);

insert into libros(titulo,autor,codigoeditorial,precio)
values('Aprenda PHP','Mario Molina',2,50);

insert into libros(titulo,autor,codigoeditorial,precio)
values('Java en 10 minutos',default,4,45);

-- Realizamos una combinación izquierda para obtener los datos
-- de los libros, incluyendo el nombre de la editorial:

```

```
select titulo,nombre  
from editoriales as e  
left join libros as l  
on codigoeditorial = e.codigo;  
-- Las editoriales de las cuales no hay libros, es decir, cuyo código de  
-- editorial no está presente en "libros" aparece en el resultado,  
-- pero con el valor "null" en el campo "titulo".
```

-- Realizamos la misma consulta anterior pero cambiamos el orden de las tablas:

```
select titulo,nombre  
from libros as l  
left join editoriales as e  
on codigoeditorial = e.codigo;  
-- El resultado mostrará el título del libro y el nombre de la editorial;  
-- los títulos cuyo código de editorial no está presente en "editoriales"  
-- aparecen en el resultado, pero con el valor "null" en el campo "nombre".
```

-- Restringimos el resultado de una consulta considerando solamente los registros

-- que encuentran coincidencia en la tabla de la derecha,
-- es decir, cuyo valor de código está presente en "libros":

```
select titulo,nombre  
from editoriales as e  
left join libros as l  
on e.codigo=codigoeditorial  
where codigoeditorial is not null;
```

-- Mostramos las editoriales que NO están presentes en "libros",

-- es decir, que NO encuentran coincidencia en la tabla de la derecha:

```
select titulo,nombre
```

```
from editoriales as e  
left join libros as l  
on e.codigo=codigoeditorial  
where codigoeditorial is null;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL query editor window titled "bd1 on postgres@PostgreSQL 11". The query is as follows:

```
30  
31 -- Realizamos una combinación izquierda para obtener los datos  
32 -- de los libros, incluyendo el nombre de la editorial:  
33 select titulo,nombre  
34 from editoriales as e  
35 left join libros as l  
36 on codigoeditorial = e.codigo;  
37 -- Las editoriales de las cuales no hay libros, es decir, cuyo código de  
38 -- editorial no está presente en "libros" aparece en el resultado,  
39 -- pero con el valor "null" en el campo "titulo",  
40
```

The results are displayed in a table:

	titulo	nombre
1	El eleph	Planeta
2	Martin Fierro	Planeta
3	Aprenda PHP	Emece
4	[null]	Siglo XXI

46 - Combinación externa derecha (right join)

Vimos que una combinación externa izquierda (left join) encuentra registros de la tabla izquierda que se correspondan con los registros de la tabla derecha y si un valor de la tabla izquierda no se encuentra en la tabla derecha, el registro muestra los campos correspondientes a la tabla de la derecha seteados a "null".

Una combinación externa derecha ("right outer join" o "right join") opera del mismo modo sólo que la tabla derecha es la que localiza los registros en la tabla izquierda.

En el siguiente ejemplo solicitamos el título y nombre de la editorial de los libros empleando un "right join":

```
select titulo,nombre  
from libros as l  
right join editoriales as e  
on codigoeditorial = e.codigo;
```

El resultado mostrará el título y nombre de la editorial; las editoriales de las cuales no hay libros, es decir, cuyo código de editorial no está presente en "libros" aparece en el resultado, pero con el valor "null" en el campo "titulo".

Es FUNDAMENTAL tener en cuenta la posición en que se colocan las tablas en los "outer join". En un "left join" la primera tabla (izquierda) es la que busca coincidencias en la segunda tabla (derecha); en el "right join" la segunda tabla (derecha) es la que busca coincidencias en la primera tabla (izquierda).

En la siguiente consulta empleamos un "left join" para conseguir el mismo resultado que el "right join" anterior:

```
select titulo,nombre  
from editoriales as e  
left join libros as l  
on codigoeditorial = e.codigo;
```

Note que la tabla que busca coincidencias ("editoriales") está en primer lugar porque es un "left join"; en el "right join" precedente, estaba en segundo lugar.

Un "right join" hace coincidir registros en una tabla (derecha) con otra tabla (izquierda); si un valor de la tabla de la derecha no encuentra coincidencia en la tabla izquierda, se genera una fila extra (una por cada valor no encontrado) con todos los campos correspondientes a la tabla izquierda seteados a "null". La sintaxis básica es la siguiente:

```
select CAMPOS  
from TABLAIZQUIERDA  
right join TABLADERECHA  
on CONDICION;
```

Un "right join" también puede tener cláusula "where" que restrinja el resultado de la consulta considerando solamente los registros que encuentran coincidencia en la tabla izquierda:

```
select titulo,nombre  
from libros as l  
right join editoriales as e  
on e.codigo=codigoeditorial  
where codigoeditorial is not null;
```

Mostramos las editoriales que NO están presentes en "libros", es decir, que NO encuentran coincidencia en la tabla de la derecha empleando un "right join":

```
select titulo,nombre  
from libros as l  
rightjoin editoriales as e  
on e.codigo=codigoeditorial  
where codigoeditorial is null;
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;  
drop table if exists editoriales;  
  
create table libros(  
codigo serial,  
titulo varchar(40),  
autor varchar(30) default 'Desconocido',  
codigoeditorial smallint not null,  
precio decimal(5,2),  
primary key(codigo)
```

```

);

create table editoriales(
    codigo serial,
    nombre varchar(20),
    primary key (codigo)
);

insert into editoriales(nombre) values('Planeta');

insert into editoriales(nombre) values('Emece');

insert into editoriales(nombre) values('Siglo XXI');

insert into libros(titulo,autor,codigoeditorial,precio)
values('El aleph','Borges',1,20);

insert into libros(titulo,autor,codigoeditorial,precio)
values('Martin Fierro','Jose Hernandez',1,30);

insert into libros(titulo,autor,codigoeditorial,precio)
values('Aprenda PHP','Mario Molina',2,50);

insert into libros(titulo,autor,codigoeditorial,precio)
values('Java en 10 minutos',default,4,45);

-- Solicitamos el título y nombre de la editorial de los libros empleando
-- un "right join":
select titulo,nombre
from libros as l
right join editoriales as e
on codigoeditorial = e.codigo;

-- Las editoriales de las cuales no hay libros, es decir, cuyo código de editorial
-- no está presente en "libros" aparece en el resultado, pero con el valor
-- "null" en el campo "titulo".

```

-- Realizamos la misma consulta anterior agregando un "where" que restrinja el resultado considerando solamente los registros que encuentran coincidencia en la tabla izquierda:

```
select titulo,nombre  
from libros as l  
right join editoriales as e  
on e.codigo=codigoeditorial  
where codigoeditorial is not null;
```

-- Mostramos las editoriales que NO están presentes en "libros"

-- (que NO encuentran coincidencia en "editoriales"):

```
select titulo,nombre  
from libros as l  
right join editoriales as e  
on e.codigo=codigoeditorial  
where codigoeditorial is null;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL 11  
38  
39 -- Solicitamos el título y nombre de la editorial de los libros empleando  
39 -- un "right join":  
40 select titulo,nombre  
41 from libros as l  
42 right join editoriales as e  
43 on codigoeditorial = e.codigo;  
44 -- Las editoriales de las cuales no hay libros, es decir, cuyo código de editorial  
45 -- no está presente en "libros" aparece en el resultado, pero con el valor  
46 -- "null" en el campo "titulo".  
47
```

	titulo	nombre
1	El aleph	Planeta
2	Martin Fierro	Planeta
3	Aprenda PHP	Emece
4	[null]	Siglo XXI

47 - Combinación externa completa (full join)

Vimos que un "left join" encuentra registros de la tabla izquierda que se correspondan con los registros de la tabla derecha y si un valor de la tabla izquierda no se encuentra en la tabla derecha, el registro muestra los campos correspondientes a la tabla de la derecha seteados a "null". Aprendimos también que un "right join" opera del mismo modo sólo que la tabla derecha es la que localiza los registros en la tabla izquierda.

Una combinación externa completa ("full outer join" o "full join") retorna todos los registros de ambas tablas. Si un registro de una tabla izquierda no encuentra coincidencia en la tabla derecha, las columnas correspondientes a campos de la tabla derecha aparecen seteadas a "null", y si la tabla de la derecha no encuentra correspondencia en la tabla izquierda, los campos de esta última aparecen conteniendo "null".

Veamos un ejemplo:

```
select titulo,nombre  
from editoriales as e  
full join libros as l  
on codigoeditorial = e.codigo;
```

La salida del "full join" precedente muestra todos los registros de ambas tablas, incluyendo los libros cuyo código de editorial no existe en la tabla "editoriales" y las editoriales de las cuales no hay correspondencia en "libros".

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;  
drop table if exists editoriales;  
  
create table libros(  
    codigo serial,  
    titulo varchar(40),  
    autor varchar(30) default 'Desconocido',  
    codigoeditorial smallint not null,  
    precio decimal(5,2),  
    primary key(codigo)
```

```

);

create table editoriales(
    codigo serial,
    nombre varchar(20),
    primary key (codigo)

);

insert into editoriales(nombre) values('Planeta');

insert into editoriales(nombre) values('Emece');

insert into editoriales(nombre) values('Siglo XXI');

insert into libros(titulo,autor,codigoeditorial,precio)
values('El aleph','Borges',1,20);

insert into libros(titulo,autor,codigoeditorial,precio)
values('Martin Fierro','Jose Hernandez',1,30);

insert into libros(titulo,autor,codigoeditorial,precio)
values('Aprenda PHP','Mario Molina',2,50);

insert into libros(titulo,autor,codigoeditorial,precio)
values('Java en 10 minutos',default,4,45);

```

-- Realizamos una combinación externa completa para obtener todos los
-- registros de ambas tablas, incluyendo los libros cuyo código de
-- editorial no existe en la tabla "editoriales"
-- y las editoriales de las cuales no hay correspondencia en "libros":

```

select titulo,nombre
from editoriales as e
full join libros as l
on codigoeditorial = e.codigo;

```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL 11
31: -- Realizamos una combinación externa completa para obtener todos los
32: -- registros de ambas tablas, incluyendo los libros cuyo código de
33: -- editorial no existe en la tabla "editoriales"
34: -- y las editoriales de las cuales no hay correspondencia en "libros":
35: select titulo,nombre
36:   from editoriales as e
37: full join libros as l
38:    on codigoeditorial = e.codigo;
39:
40:
```

Data Output		Explain	Messages	Notifications	Query History
	titulo character varying (40)	nombre character varying (20)			
1	El aleph	Planeta			
2	Martin Fierro	Planeta			
3	Aprenda PHP	Emece			
4	Java en 10 minutos	[null]			
5	[null]	Siglo XXI			

48 - Combinaciones cruzadas (cross join)

Vimos que hay tres tipos de combinaciones: 1) combinaciones internas (join), 2) combinaciones externas (left, right y full join) y 3) combinaciones cruzadas.

Las combinaciones cruzadas (cross join) muestran todas las combinaciones de todos los registros de las tablas combinadas. Para este tipo de join no se incluye una condición de enlace. Se genera el producto cartesiano en el que el número de filas del resultado es igual al número de registros de la primera tabla multiplicado por el número de registros de la segunda tabla, es decir, si hay 5 registros en una tabla y 6 en la otra, retorna 30 filas.

La sintaxis básica es ésta:

```
select CAMPOS  
from TABLA1  
cross join TABLA2;
```

Veamos un ejemplo. Un pequeño restaurante almacena los nombres y precios de sus comidas en una tabla llamada "comidas" y en una tabla denominada "postres" los mismos datos de sus postres.

Si necesitamos conocer todas las combinaciones posibles para un menú, cada comida con cada postre, empleamos un "cross join":

```
select c.nombre as platoprincipal, p.nombre as postre  
from comidas as c  
cross join postres as p;
```

La salida muestra cada plato combinado con cada uno de los postres.

Como cualquier tipo de "join", puede emplearse una cláusula "where" que condicione la salida.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists comidas;  
drop table if exists postres;
```

```
create table comidas(  
codigo serial,  
nombre varchar(30),  
precio decimal(4,2),
```

```

primary key(codigo)
);

create table postres(
    codigo serial,
    nombre varchar(30),
    precio decimal(4,2),
    primary key(codigo)
);

insert into comidas(nombre,precio) values('ravioles',5);
insert into comidas(nombre,precio) values('tallarines',4);
insert into comidas(nombre,precio) values('milanesa',7);
insert into comidas(nombre,precio) values('cuarto de pollo',6);

insert into postres(nombre,precio) values('flan',2.5);
insert into postres(nombre,precio) values('porcion torta',3.5);

-- El restaurante quiere combinar los registros de ambas tablas
-- para mostrar los distintos menúes que ofrece.
-- Lo hacemos usando un "cross join":
select c.nombre as platoprincipal,
       p.nombre as postre,
       c.precio+p.precio as total
  from comidas as c
 cross join postres as p;

```

La ejecución de este lote de comandos SQL genera una salida similar a:

```

25 bd1 on postgres@PostgreSQL11
26 -- El restaurante quiere combinar los registros de ambas tablas
27 -- para mostrar los distintos menúes que ofrece.
28 -- Lo hacemos usando un "cross join":
29 select c.nombre as platoprincipal,
30       p.nombre as postre,
31       c.precio+p.precio as total
32   from comidas as c
33 cross join postres as p;
34

```

	Data Output	Explain	Messages	Notifications	Query History
#	platoprincipal character varying (30)	postre character varying (30)	total numeric		
1	ravioles	flan	7.50		
2	ravioles	porcion torta	8.50		
3	tallarines	flan	6.50		
4	tallarines	porcion torta	7.50		
5	milanesa	flan	9.50		
6	milanesa	porcion torta	10.50		
7	cuarto de pollo	flan	8.50		
8	cuarto de pollo	porcion torta	9.50		

49 - Autocombinación

Dijimos que es posible combinar una tabla consigo misma.

Un pequeño restaurante tiene almacenadas sus comidas en una tabla llamada "comidas" que consta de los siguientes campos:

- nombre varchar(20),
- precio decimal (4,2) y
- rubro char(6)-- que indica con 'plato' si es un plato principal y 'postre' si es postre.

Podemos obtener la combinación de platos empleando un "cross join" con una sola tabla:

```
select c1.nombre as platoprincipal,  
       c2.nombre as postre,  
       c1.precio+c2.precio as total  
  from comidas as c1  
 cross join comidas as c2;
```

En la consulta anterior aparecen filas duplicadas, para evitarlo debemos emplear un "where":

```
select c1.nombre as platoprincipal,  
       c2.nombre as postre,  
       c1.precio+c2.precio as total  
  from comidas as c1  
 cross join comidas as c2  
 where c1.rubro='plato' and  
       c2.rubro='postre';
```

En la consulta anterior se empleó un "where" que especifica que se combine "plato" con "postre".

En una autocombinación se combina una tabla con una copia de si misma. Para ello debemos utilizar 2 alias para la tabla. Para evitar que aparezcan filas duplicadas, debemos emplear un "where".

También se puede realizar una autocombinación con "join":

```
select c1.nombre as platoprincipal,
```

```

c2.nombre as postre,
c1.precio+c2.precio as total

from comidas as c1

join comidas as c2

on c1.codigo=>c2.codigo

where c1.rubro='plato' and

c2.rubro='postre';

```

Para que no aparezcan filas duplicadas se agrega un "where".

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists comidas;
```

```

create table comidas(
codigo serial,
nombre varchar(30),
precio decimal(4,2),
rubro char(6),-- 'plato'=plato principal, 'postre'=postre
primary key(codigo)
);

```

```

insert into comidas(nombre,precio,rubro) values('ravioles',5,'plato');

insert into comidas(nombre,precio,rubro) values('tallarines',4,'plato');

insert into comidas(nombre,precio,rubro) values('milanesa',7,'plato');

insert into comidas(nombre,precio,rubro) values('cuarto de pollo',6,'plato');

insert into comidas(nombre,precio,rubro) values('flan',2.5,'postre');

insert into comidas(nombre,precio,rubro) values('porcion torta',3.5,'postre');

```

-- Realizamos un "cross join":

```

select c1.nombre as platoprincipal,
c2.nombre as postre,

```

```
c1.precio+c2.precio as total
```

```
from comidas as c1
```

```
cross join comidas as c2;
```

-- Note que aparecen filas duplicadas, por ejemplo, "ravioles" se combina con

-- "ravioles" y la combinación "ravioles- flan" se repite como "flan- ravioles".

-- Debemos especificar que combine el rubro "plato" con "postre":

```
select c1.nombre as platoprincipal,
```

```
c2.nombre as postre,
```

```
c1.precio+c2.precio as total
```

```
from comidas as c1
```

```
cross join comidas as c2
```

```
where c1.rubro='plato' and
```

```
c2.rubro='postre';
```

-- La salida muestra cada plato combinado con cada postre,

-- y una columna extra que calcula el total del menú.

-- También se puede realizar una autocombinación con "join":

```
select c1.nombre as platoprincipal,
```

```
c2.nombre as postre,
```

```
c1.precio+c2.precio as total
```

```
from comidas as c1
```

```
join comidas as c2
```

```
on c1.codigo<>c2.codigo
```

```
where c1.rubro='plato' and
```

```
c2.rubro='postre';
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```

bdt on postgres@PostgreSQL 11

25  -- "ravioles" y la combinación "ravioles- flan" se repite como "flan- ravioles".
26
27  -- Debemos especificar que combine el rubro "plato" con "postre":
28  select c1.nombre as platoprincipal,
29    c2.nombre as postre,
30    c1.precio+c2.precio as total
31  from comidas as c1
32  cross join comidas as c2
33  where c1.rubro='plato' and
34    c2.rubro='postre';
35  -- La salida muestra cada plato combinado con cada postre,
36  -- y una columna extra que calcula el total del menú.
37

```

Data Output Explain Messages Notifications Query History

	platoprincipal character varying (30)	postre character varying (30)	total numeric
1	ravioles	flan	7.50
2	ravioles	porcion torta	8.50
3	tallarines	flan	6.50
4	tallarines	porcion torta	7.50
5	milanesa	flan	9.50
6	milanesa	porcion torta	10.50
7	cuarto de pollo	flan	8.50
8	cuarto de pollo	porcion torta	9.50

50 - Combinaciones y funciones de agrupamiento

Podemos usar "group by" y las funciones de agrupamiento con combinaciones de tablas.

Para ver la cantidad de libros de cada editorial consultando la tabla "libros" y "editoriales", tipeamos:

```
select nombre as editorial,  
       count(*) as cantidad  
  from editoriales as e  
  join libros as l  
    on codigoeditorial=e.codigo  
 group by e.nombre;
```

Note que las editoriales que no tienen libros no aparecen en la salida porque empleamos un "join".

Empleamos otra función de agrupamiento con "left join". Para conocer el mayor precio de los libros de cada editorial usamos la función "max()", hacemos un "left join" y agrupamos por nombre de la editorial:

```
select nombre as editorial,  
       max(precio) as mayorprecio  
  from editoriales as e  
 left join libros as l  
    on codigoeditorial=e.codigo  
 group by nombre;
```

En la sentencia anterior, mostrará, para la editorial de la cual no haya libros, el valor "null" en la columna calculada.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;  
drop table if exists editoriales;  
  
create table libros(
```

```

codigo serial,
titulo varchar(40),
autor varchar(30),
codigoeditorial smallint not null,
precio decimal(5,2),
primary key(codigo)

);

create table editoriales(
codigo serial,
nombre varchar(20),
primary key (codigo)

);

insert into editoriales(nombre) values('Planeta');

insert into editoriales(nombre) values('Emece');

insert into editoriales(nombre) values('Siglo XXI');

insert into libros(titulo,autor,codigoeditorial,precio)
values('El aleph','Borges',1,20);

insert into libros(titulo,autor,codigoeditorial,precio)
values('Martin Fierro','Jose Hernandez',1,30);

insert into libros(titulo,autor,codigoeditorial,precio)
values('Aprenda PHP','Mario Molina',3,50);

insert into libros(titulo,autor,codigoeditorial,precio)
values('Uno','Richard Bach',3,15);

insert into libros(titulo,autor,codigoeditorial,precio)
values('Java en 10 minutos',default,4,45);

```

-- Contamos la cantidad de libros de cada editorial consultando ambas tablas:

```
select nombre as editorial,
       count(*) as cantidad
  from editoriales as e
 join libros as l
    on codigoeditorial=e.codigo
 group by e.nombre;
```

-- Buscamos el libro más costoso de cada editorial con un "left join":

```
select nombre as editorial,
       max(precio) as mayorprecio
  from editoriales as e
 left join libros as l
    on codigoeditorial=e.codigo
 group by nombre;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL 11
40
41 -- Buscamos el libro más costoso de cada editorial con un "left join":
42 select nombre as editorial,
43        max(precio) as mayorprecio
44   from editoriales as e
45  left join libros as l
46    on codigoeditorial=e.codigo
47   group by nombre;
48
```

Data Output	
editorial character varying (20)	mayorprecio numeric
1 Emece	[null]
2 Planeta	30.00
3 Siglo XXI	50.00

51 - Combinación de más de dos tablas

Podemos hacer un "join" con más de dos tablas.

Cada join combina 2 tablas. Se pueden emplear varios join para enlazar varias tablas. Cada resultado de un join es una tabla que puede combinarse con otro join.

La librería almacena los datos de sus libros en tres tablas: libros, editoriales y autores.

En la tabla "libros" un campo "codigoautor" hace referencia al autor y un campo "codigoeditorial" referencia la editorial.

Para recuperar todos los datos de los libros empleamos la siguiente consulta:

```
select titulo,a.nombre,e.nombre  
from autores as a  
join libros as l  
on codigoautor=a.codigo  
join editoriales as e  
on codigoeditorial=e.codigo;
```

Analicemos la consulta anterior. Indicamos el nombre de la tabla luego del "from" ("autores"), combinamos esa tabla con la tabla "libros" especificando con "on" el campo por el cual se combinan; luego debemos hacer coincidir los valores para el enlace con la tabla "editoriales" enlazándolas por los campos correspondientes. Utilizamos alias para una sentencia más sencilla y comprensible.

Note que especificamos a qué tabla pertenecen los campos cuyo nombre se repiten en las tablas, esto es necesario para evitar confusiones y ambigüedades al momento de referenciar un campo.

Note que no aparecen los libros cuyo código de autor no se encuentra en "autores" y cuya editorial no existe en "editoriales", esto es porque realizamos una combinación interna.

Podemos combinar varios tipos de join en una misma sentencia:

```
select titulo,a.nombre,e.nombre  
from autores as a  
right join libros as l  
on codigoautor=a.codigo  
left join editoriales as e on codigoeditorial=e.codigo;
```

En la consulta anterior solicitamos el título, autor y editorial de todos los libros que encuentren o no coincidencia con "autores" ("right join") y a ese resultado lo combinamos con "editoriales", encuentren o no coincidencia.

Es posible realizar varias combinaciones para obtener información de varias tablas. Las tablas deben tener claves externas relacionadas con las tablas a combinar.

En consultas en las cuales empleamos varios "join" es importante tener en cuenta el orden de las tablas y los tipos de "join"; recuerde que la tabla resultado del primer join es la que se combina con el segundo join, no la segunda tabla nombrada. En el ejemplo anterior, el "left join" no se realiza entre las tablas "libros" y "editoriales" sino entre el resultado del "right join" y la tabla "editoriales".

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
drop table if exists autores;
```

```
drop table if exists editoriales;
```

```
create table libros(
```

```
    codigo serial,
```

```
    titulo varchar(40),
```

```
    codigoautor int not null,
```

```
    codigoeditorial smallint not null,
```

```
    precio decimal(5,2),
```

```
    primary key(codigo)
```

```
);
```

```
create table autores(
```

```
    codigo serial,
```

```
    nombre varchar(20),
```

```
    primary key (codigo)
```

```
);
```

```
create table editoriales(
```

```
    codigo serial,
```

```
    nombre varchar(20),
```

```
    primary key (codigo)
```

```

);

insert into editoriales(nombre) values('Planeta');

insert into editoriales(nombre) values('Emece');

insert into editoriales(nombre) values('Siglo XXI');

insert into editoriales(nombre) values('Plaza');

insert into autores(nombre) values ('Richard Bach');

insert into autores(nombre) values ('Borges');

insert into autores(nombre) values ('Jose Hernandez');

insert into autores(nombre) values ('Mario Molina');

insert into autores(nombre) values ('Paenza');

insert into libros(titulo,codigoautor,codigoeditorial,precio)

values('El aleph',2,2,20);

insert into libros(titulo,codigoautor,codigoeditorial,precio)

values('Martin Fierro',3,1,30);

insert into libros(titulo,codigoautor,codigoeditorial,precio)

values('Aprenda PHP',4,3,50);

insert into libros(titulo,codigoautor,codigoeditorial,precio)

values('Uno',1,1,15);

insert into libros(titulo,codigoautor,codigoeditorial,precio)

values('Java en 10 minutos',0,3,45);

insert into libros(titulo,codigoautor,codigoeditorial,precio)

values('Matematica estas ahí',0,0,15);

insert into libros(titulo,codigoautor,codigoeditorial,precio)

values('Java de la A a la Z',4,0,50);

```

-- Recuperamos todos los datos de los libros consultando las tres tablas:

```
select titulo,a.nombre,e.nombre,precio
```

```

from autores as a
join libros as l
on codigoautor=a.codigo
join editoriales as e
on codigoeditorial=e.codigo;
-- Los libros cuyo código de autor no se encuentra en "autores"
-- (caso de "Java en 10 minutos" y "Matematica estas ahí") y
-- cuya editorial no existe en "editoriales"
-- (caso de "Matematica estas ahí" y "Java de la A a la Z"),
-- no aparecen porque realizamos una combinación interna.

```

-- Podemos combinar varios tipos de join en una misma sentencia:

```
select titulo,a.nombre,e.nombre,precio
```

```
from autores as a
```

```
right join libros as l
```

```
on codigoautor=a.codigo
```

```
left join editoriales as e
```

```
on codigoeditorial=e.codigo;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```

51
52 -- Recuperamos todos los datos de los libros consultando las tres tablas:
53 select titulo,a.nombre,e.nombre,precio
54   from autores as a
55   join libros as l
56   on codigoautor=a.codigo
57   join editoriales as e
58   on codigoeditorial=e.codigo;
59 -- Los libros cuyo código de autor no se encuentra en "autores"
60 -- (caso de "Java en 10 minutos" y "Matematica estas ahí") y
61 -- cuya editorial no existe en "editoriales"
62 -- (caso de "Matematica estas ahí" y "Java de la A a la Z"),
63 -- no aparecen porque realizamos una combinación interna.
64

```

Data Output Explain Messages Notifications Query History				
	titulo character varying (40)	nombre character varying (20)	nombre character varying (20)	precio numeric (5,2)
1	El aleph	Borges	Emece	20.00
2	Martin Fierro	Jose Hernandez	Planeta	30.00
3	Aprenda PHP	Mario Molina	Siglo XXI	50.00
4	Uno	Richard Bach	Planeta	15.00

52 - Clave foránea

Un campo que no es clave primaria en una tabla y sirve para enlazar sus valores con otra tabla en la cual es clave primaria se denomina clave foránea, externa o ajena.

En el ejemplo de la librería en que utilizamos las tablas "libros" y "editoriales" con estos campos:

libros: codigo (clave primaria), titulo, autor, codigoeditorial, precio y

editoriales: codigo (clave primaria), nombre.

el campo "codigoeditorial" de "libros" es una clave foránea, se emplea para enlazar la tabla "libros" con "editoriales" y es clave primaria en "editoriales" con el nombre "codigo".

Las claves foráneas y las claves primarias deben ser del mismo tipo para poder enlazarse. Si modificamos una, debemos modificar la otra para que los valores se correspondan.

Cuando alteramos una tabla, debemos tener cuidado con las claves foráneas. Si modificamos el tipo, longitud o atributos de una clave foránea, ésta puede quedar inhabilitada para hacer los enlaces.

Entonces, una clave foránea es un campo (o varios) empleados para enlazar datos de 2 tablas, para establecer un "join" con otra tabla en la cual es clave primaria.

53 - Restricciones (foreign key)

Hemos visto que una de las alternativas que PostgreSQL ofrece para asegurar la integridad de datos es el uso de restricciones (constraints). Aprendimos que las restricciones se establecen en tablas y campos asegurando que los datos sean válidos y que las relaciones entre las tablas se mantengan.

Con la restricción "foreign key" se define un campo (o varios) cuyos valores coinciden con la clave primaria de la misma tabla o de otra, es decir, se define una referencia a un campo con una restricción "primary key" o "unique" de la misma tabla o de otra.

La integridad referencial asegura que se mantengan las referencias entre las claves primarias y las externas. Por ejemplo, controla que si se agrega un código de editorial en la tabla "libros", tal código exista en la tabla "editoriales".

También controla que no pueda eliminarse un registro de una tabla ni modificar la clave primaria si una clave externa hace referencia al registro. Por ejemplo, que no se pueda eliminar o modificar un código de "editoriales" si existen libros con dicho código.

La siguiente es la sintaxis parcial general para agregar una restricción "foreign key":

```
alter table NOMBRERESTRICCION  
add constraint NOMBRERESTRICCION  
foreign key (CAMPOCLAVEFORANEA)  
references NOMBRERESTRICCION (CAMPOCLAVEPRIMARIA);
```

Analicémosla:

- NOMBRERESTRICCION referencia el nombre de la tabla a la cual le aplicamos la restricción,
- NOMBRERESTRICCION es el nombre que le damos a la misma,
- luego de "foreign key", entre paréntesis se coloca el campo de la tabla a la que le aplicamos la restricción que será establecida como clave foránea,
- luego de "references" indicamos el nombre de la tabla referenciada y el campo que es clave primaria en la misma, a la cual hace referencia la clave foránea. La tabla referenciada debe tener definida una restricción "primary key" o "unique"; si no la tiene, aparece un mensaje de error.

Para agregar una restricción "foreign key" al campo "codigoeditorial" de "libros", tipeamos:

```
alter table libros  
add constraint FK_libros_codigoeditorial  
foreign key (codigoeditorial)
```

references editoriales(codigo);

En el ejemplo implementamos una restricción "foreign key" para asegurarnos que el código de la editorial de la tabla "libros" ("codigoeditorial") esté asociada con un código válido en la tabla "editoriales" ("codigo").

Cuando agregamos cualquier restricción a una tabla que contiene información, PostgreSQL controla los datos existentes para confirmar que cumplen con la restricción, si no los cumple, la restricción no se aplica y aparece un mensaje de error. Por ejemplo, si intentamos agregar una restricción "foreign key" a la tabla "libros" y existe un libro con un valor de código para editorial que no existe en la tabla "editoriales", la restricción no se agrega.

Actúa en inserciones. Si intentamos ingresar un registro (un libro) con un valor de clave foránea (codigoeditorial) que no existe en la tabla referenciada (editoriales), PostgreSQL muestra un mensaje de error. Si al ingresar un registro (un libro), no colocamos el valor para el campo clave foránea (codigoeditorial), almacenará "null", porque esta restricción permite valores nulos (a menos que se haya especificado lo contrario al definir el campo).

Actúa en eliminaciones y actualizaciones. Si intentamos eliminar un registro o modificar un valor de clave primaria de una tabla si una clave foránea hace referencia a dicho registro, PostgreSQL no lo permite. Por ejemplo, si intentamos eliminar una editorial a la que se hace referencia en "libros", aparece un mensaje de error.

Esta restricción (a diferencia de "primary key" y "unique") no crea índice automáticamente.

La cantidad y tipo de datos de los campos especificados luego de "foreign key" DEBEN coincidir con la cantidad y tipo de datos de los campos de la cláusula "references".

Una tabla puede tener varias restricciones "foreign key".

No se puede eliminar una tabla referenciada en una restricción "foreign key", aparece un mensaje de error.

Una restriccion "foreign key" no puede modificarse, debe eliminarse y volverse a crear.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
drop table if exists editoriales;
```

```
create table libros(
```

```
    codigo serial,
```

```
    titulo varchar(40),
```

```
    autor varchar(30),
```

```
    codigoeditorial smallint,
```

```
primary key(codigo)
);

create table editoriales(
    codigo serial,
    nombre varchar(20),
    primary key (codigo)
);
```

```
insert into editoriales(nombre) values('Emece');
insert into editoriales(nombre) values('Planeta');
insert into editoriales(nombre) values('Siglo XXI');
```

```
insert into libros(titulo,autor,codigoeditorial) values('El aleph','Borges',1);
insert into libros(titulo,autor,codigoeditorial) values('Martin Fierro','Jose Hernandez',2);
insert into libros(titulo,autor,codigoeditorial) values('Aprenda PHP','Mario Molina',2);
```

-- Agregamos una restricción "foreign key" a la tabla "libros":

```
alter table libros
add constraint FK_libros_codigoeditorial
foreign key (codigoeditorial)
references editoriales(codigo);
```

-- Ingresamos un libro con un código de editorial existente:

```
insert into libros(titulo,autor,codigoeditorial) values('Aprenda ASP.Net','Jose Paez',2);
```

-- Ingresamos un libro con un código de editorial inexistente:

```
insert into libros(titulo,autor,codigoeditorial) values('JSP basico','Tornado Luis',7);
```

--Aparece un mensaje de error y no se ejecuta la inserción.

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL:11
23
24
25 -- Agregamos una restricción "foreign key" a la tabla "libros";
26 alter table libros
27   add constraint FK_libros_codigoeditorial
28     foreign key (codigoeditorial)
29       references editoriales(codigo);
30
31 -- Ingresamos un libro con un código de editorial existente;
32 insert into libros(titulo,autor,codigoeditorial) values('Aprende PHP','Hernando Mochis',2);
33
34 -- Ingresamos un libro con un código de editorial inexistente;
35 insert into libros(titulo,autor,codigoeditorial) values('JSP basico','Tornado Luis',7);
36 --Aparece un mensaje de error y no se ejecuta la inserción.
```

Data Output Explain Messages Notifications Query History

ERROR: inserción o actualización en la tabla «libros» viola la llave foránea «fk_libros_codigoeditorial»
DETAIL: La llave (codigoeditorial)=(7) no está presente en la tabla «editoriales».
SQL state: 23503

54 - Restricciones foreign key en la misma tabla

La restricción "foreign key", que define una referencia a un campo con una restricción "primary key" o "unique" se puede definir entre distintas tablas (como hemos aprendido) o dentro de la misma tabla.

Veamos un ejemplo en el cual definimos esta restricción dentro de la misma tabla.

Una mutual almacena los datos de sus afiliados en una tabla llamada "afiliados". Algunos afiliados inscriben a sus familiares. La tabla contiene un campo que hace referencia al afiliado que lo incorporó a la mutual, del cual dependen.

La estructura de la tabla es la siguiente:

```
create table afiliados(  
    numero serial,  
    documento char(8) not null,  
    nombre varchar(30),  
    afiliadotitular int,  
    primary key (documento),  
    unique (numero)  
);
```

En caso que un afiliado no haya sido incorporado a la mutual por otro afiliado, el campo "afiliadotitular" almacenará "null".

Establecemos una restricción "foreign key" para asegurarnos que el número de afiliado que se ingrese en el campo "afiliadotitular" exista en la tabla "afiliados":

```
alter table afiliados  
add constraint FK_afiliados_afiliadotitular  
foreign key (afiliadotitular)  
references afiliados (numero);
```

La sintaxis es la misma, excepto que la tabla se autoreferencia.

Luego de aplicar esta restricción, cada vez que se ingrese un valor en el campo "afiliadotitular", PostgreSQL controlará que dicho número exista en la tabla, si no existe, mostrará un mensaje de error.

Si intentamos eliminar un afiliado que es titular de otros afiliados, no se podrá hacer.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists afiliados;
```

```
create table afiliados(  
    numero serial,  
    documento char(8) not null,  
    nombre varchar(30),  
    afiliadotitular int,  
    primary key (documento),  
    unique (numero)  
);
```

```
-- Establecemos una restricción "foreign key" para asegurarnos que el número de afiliado  
-- que se ingrese en el campo "afiliadotitular" exista en la tabla "afiliados":
```

```
alter table afiliados  
add constraint FK_afiliados_afiliadotitular  
foreign key (afiliadotitular)  
references afiliados (numero);
```

```
-- Ingresamos algunos registros:
```

```
insert into afiliados(documento,nombre,afiliadotitular) values('22222222','Perez Juan',null);  
insert into afiliados(documento,nombre,afiliadotitular) values('23333333','Garcia Maria',null);  
insert into afiliados(documento,nombre,afiliadotitular) values('24444444','Lopez Susana',null);  
insert into afiliados(documento,nombre,afiliadotitular) values('30000000','Perez Marcela',1);  
insert into afiliados(documento,nombre,afiliadotitular) values('31111111','Morales Luis',1);  
insert into afiliados(documento,nombre,afiliadotitular) values('32222222','Garcia Maria',2);
```

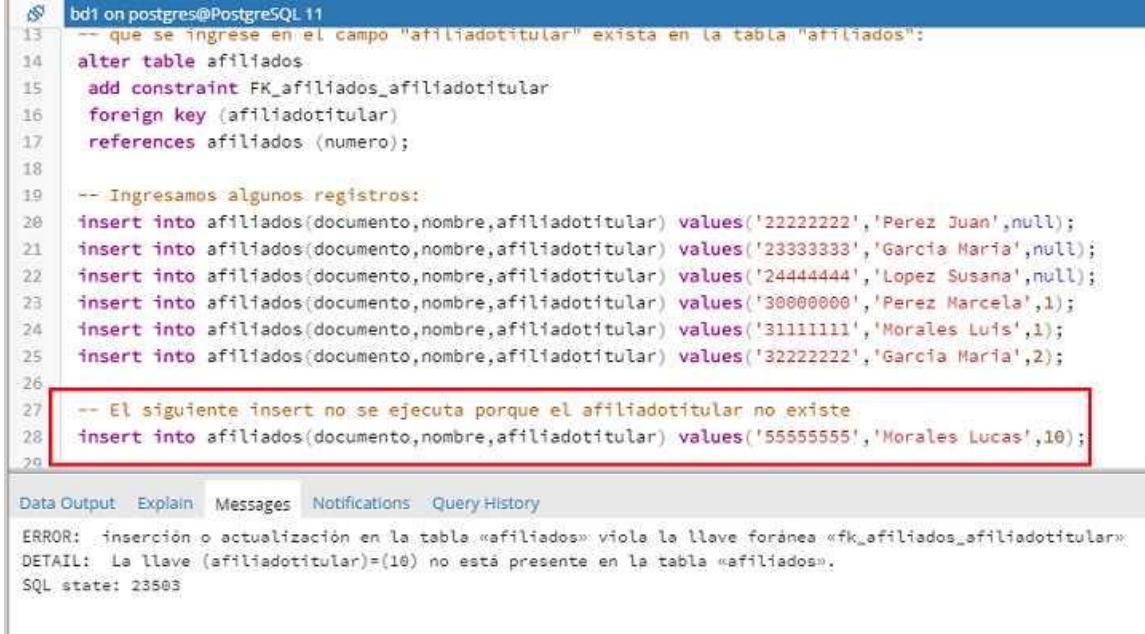
```
-- El siguiente insert no se ejecuta porque el afiliadotitular no existe
```

```
insert into afiliados(documento,nombre,afiliadotitular) values('55555555','Morales Lucas',10);
```

- Podemos eliminar un afiliado, siempre que no haya otro afiliado que haga referencia a él
- en "afiliadotitular", es decir, si el "numero" del afiliado está presente en algún registro
- en el campo "afiliadotitular":

```
delete from afiliados where numero=1;
```

La ejecución de este lote de comandos SQL genera una salida similar a:



The screenshot shows a PostgreSQL terminal window with the following content:

```

bd1 on postgres@PostgreSQL:11
-- que se ingrese en el campo "afiliadotitular" exista en la tabla "afiliados";
13 alter table afiliados
14   add constraint FK_afiliados_afiliadotitular
15     foreign key (afiliadotitular)
16       references afiliados (numero);
17
18
19 -- Ingresamos algunos registros:
20 insert into afiliados(documento,nombre,afiliadotitular) values('22222222','Perez Juan',null);
21 insert into afiliados(documento,nombre,afiliadotitular) values('23333333','Garcia Maria',null);
22 insert into afiliados(documento,nombre,afiliadotitular) values('24444444','Lopez Susana',null);
23 insert into afiliados(documento,nombre,afiliadotitular) values('30000000','Perez Marcela',1);
24 insert into afiliados(documento,nombre,afiliadotitular) values('31111111','Morales Luis',1);
25 insert into afiliados(documento,nombre,afiliadotitular) values('32222222','Garcia Maria',2);
26
27 -- El siguiente insert no se ejecuta porque el afiliadotitular no existe
28 insert into afiliados(documento,nombre,afiliadotitular) values('55555555','Morales Lucas',10);
29

```

The terminal shows the command being run, followed by an error message:

```

Data Output Explain Messages Notifications Query History
ERROR: inserción o actualización en la tabla «afiliados» viola la llave foránea «fk_afiliados_afiliadotitular».
DETAIL: La llave (afiliadotitular)=(10) no está presente en la tabla «afiliados».
SQL state: 23503

```

A red box highlights the last two lines of the command, which failed to execute due to a foreign key constraint violation.

55 - Restricciones foreign key al crear la tabla

Hasta el momento hemos agregado restricciones a tablas existentes con "alter table" (manera aconsejada), también pueden establecerse al momento de crear una tabla (en la instrucción "create table").

En el siguiente ejemplo creamos la tabla "libros" con la restricción respectiva:

```
create table editoriales(  
    codigo serial,  
    nombre varchar(20),  
    primary key (codigo)  
);
```

```
create table libros(  
    codigo serial,  
    titulo varchar(40),  
    autor varchar(30),  
    codigoeditorial smallint references editoriales(codigo),  
    primary key(codigo)  
);
```

En el ejemplo anterior creamos una restricción "foreign key" para establecer el campo "codigoeditorial" como clave externa que haga referencia al campo "codigo" de "editoriales".

Si definimos una restricción "foreign key" al crear una tabla, la tabla referenciada debe existir.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists editoriales;  
drop table if exists libros;
```

```
create table editoriales(
    codigo serial,
    nombre varchar(20),
    primary key (codigo)
);

create table libros(
    codigo serial,
    titulo varchar(40),
    autor varchar(30),
    codigoeditorial smallint references editoriales(codigo),
    primary key(codigo)
);
```

-- Ingresamos algunos registros en ambas tablas:

```
insert into editoriales(nombre) values('Emece');
insert into editoriales(nombre) values('Planeta');
insert into editoriales(nombre) values('Siglo XXI');
```

```
insert into libros(titulo,autor,codigoeditorial) values('El aleph','Borges',1);
insert into libros(titulo,autor,codigoeditorial) values('Martin Fierro','Jose Hernandez',2);
insert into libros(titulo,autor,codigoeditorial) values('Aprenda PHP','Mario Molina',2);
```

-- Ingresamos un libro con un código de editorial inexistente:

```
insert into libros(titulo,autor,codigoeditorial) values('Aprenda ASP','Facundo Cabrera',7);
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL 11
1 drop table if exists editoriales;
2 drop table if exists libros;
3
4 create table editoriales(
5     codigo serial,
6     nombre varchar(20),
7     primary key (codigo)
8 );
9
10 create table libros(
11     codigo serial,
12     titulo varchar(40),
13     autor varchar(30),
14     codigoeditorial smallint references editoriales(codigo),
15     primary key(codigo)
16 );
17
18 -- Ingresamos algunos registros en ambas tablas:
19 insert into editoriales(nombre) values('Emece');
20 insert into editoriales(nombre) values('Planeta');
21 insert into editoriales(nombre) values('Siglo XXI');
22
23 insert into libros(titulo,autor,codigoeditorial) values('El aleph','Borges',1);
24 insert into libros(titulo,autor,codigoeditorial) values('Martin Fierro','Jose Hernandez',2);
25 insert into libros(titulo,autor,codigoeditorial) values('Aprenda PHP','Mario Molina',2);
26
27 -- Ingresamos un libro con un código de editorial inexistente:
28 insert into libros(titulo,autor,codigoeditorial) values('Aprenda ASP','Facundo Cabrera',7);
29
```

Data Output Explain Messages Notifications Query History

ERROR: Inserción o actualización en la tabla «libros» viola la llave foránea «libros_codigoeditorial_fkey».
DETAIL: La llave (codigoeditorial)=(7) no está presente en la tabla «editoriales».
SQL state: 23503

56 - Restricciones foreign key (acciones)

Continuamos con la restricción "foreign key".

Si intentamos eliminar un registro de la tabla referenciada por una restricción "foreign key" cuyo valor de clave primaria existe referenciada en la tabla que tiene dicha restricción, la acción no se ejecuta y aparece un mensaje de error. Esto sucede porque, por defecto, para eliminaciones, la opción de la restricción "foreign key" es "no action". Lo mismo sucede si intentamos actualizar un valor de clave primaria de una tabla referenciada por una "foreign key" existente en la tabla principal.

La restricción "foreign key" tiene las cláusulas "on delete" y "on update" que son opcionales. Estas cláusulas especifican cómo debe actuar PostgreSQL frente a eliminaciones y modificaciones de las tablas referenciadas en la restricción.

Las opciones para estas cláusulas son las siguientes:

- "no action": indica que si intentamos eliminar o actualizar un valor de la clave primaria de la tabla referenciada (TABLA2) que tengan referencia en la tabla principal (TABLA1), se genere un error y la acción no se realice; es la opción predeterminada.
- "cascade": indica que si eliminamos o actualizamos un valor de la clave primaria en la tabla referenciada (TABLA2), los registros coincidentes en la tabla principal (TABLA1), también se eliminan o modifiquen; es decir, si eliminamos o modificamos un valor de campo definido con una restricción "primary key" o "unique", dicho cambio se extiende al valor de clave externa de la otra tabla (integridad referencial en cascada).
- "set null": Establece con el valor null en el campo de la clave foránea.
- "set default": Establece el valor por defecto en el campo de la clave foránea.

La sintaxis completa para agregar esta restricción a una tabla es la siguiente:

```
alter table TABLA1  
add constraint NOMBRERESTRICCION  
foreign key (CAMPOCLAVEFORANEA)  
references TABLA2(CAMPOCLAVEPRIMARIA)  
on delete OPCION  
on update OPCION;
```

Sintetizando, si al agregar una restricción foreign key:

- no se especifica acción para eliminaciones (o se especifica "no_action"), y se intenta eliminar un registro de la tabla referenciada (editoriales) cuyo valor de clave primaria (codigo) existe en la tabla principal (libros), la acción no se realiza.

- se especifica "cascade" para eliminaciones ("on delete cascade") y elimina un registro de la tabla referenciada (editoriales) cuyo valor de clave primaria (codigo) existe en la tabla principal (libros), la eliminación de la tabla referenciada (editoriales) se realiza y se eliminan de la tabla principal (libros) todos los registros cuyo valor coincide con el registro eliminado de la tabla referenciada (editoriales).

- no se especifica acción para actualizaciones (o se especifica "no_action"), y se intenta modificar un valor de clave primaria (codigo) de la tabla referenciada (editoriales) que existe en el campo clave foránea (codigoeditorial) de la tabla principal (libros), la acción no se realiza.

- se especifica "cascade" para actualizaciones ("on update cascade") y se modifica un valor de clave primaria (codigo) de la tabla referenciada (editoriales) que existe en la tabla principal (libros), PostgreSQL actualiza el registro de la tabla referenciada (editoriales) y todos los registros coincidentes en la tabla principal (libros).

Veamos un ejemplo. Definimos una restricción "foreign key" a la tabla "libros" estableciendo el campo "codigoeditorial" como clave foránea que referencia al campo "codigo" de la tabla "editoriales". La tabla "editoriales" tiene como clave primaria el campo "codigo". Especificamos la acción en cascada para las actualizaciones y eliminaciones:

```
alter table libros
add constraint FK_libros_codigoeditorial
foreign key (codigoeditorial)
references editoriales(codigo)
on update cascade
on delete cascade;
```

Si luego de establecer la restricción anterior, eliminamos una editorial de "editoriales" de las cuales hay libros, se elimina dicha editorial y todos los libros de tal editorial. Y si modificamos el valor de código de una editorial de "editoriales", se modifica en "editoriales" y todos los valores iguales de "codigoeditorial" de libros también se modifican.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
drop table if exists editoriales;
```

```
create table libros(
    codigo serial,
    titulo varchar(40),
    autor varchar(30),
```

```

codigoeditorial smallint,
primary key(codigo)

);

create table editoriales(
codigo serial,
nombre varchar(20),
primary key (codigo)

);

insert into editoriales(nombre) values('Emece');

insert into editoriales(nombre) values('Planeta');

insert into editoriales(nombre) values('Siglo XXI');

insert into libros(titulo,autor,codigoeditorial) values('El aleph','Borges',1);

insert into libros(titulo,autor,codigoeditorial) values('Martin Fierro','Jose Hernandez',2);

insert into libros(titulo,autor,codigoeditorial) values('Aprenda PHP','Mario Molina',2);

-- Establecemos una restricción "foreign key" para evitar que se ingrese en "libros" un código
-- de editorial inexistente en "editoriales" con la opción "on cascade" para actualizaciones
-- y eliminaciones:

alter table libros

add constraint FK_libros_codigoeditorial
foreign key (codigoeditorial)
references editoriales(codigo)

on update cascade

on delete cascade;

-- Si actualizamos un valor de código de "editoriales", la modificación se extiende a todos
-- los registros de la tabla "libros" que hacen referencia a ella en "codigoeditorial":

```

```
update editoriales set codigo=10 where codigo=1;
```

-- Veamos si la actualización se extendió a "libros":

```
select titulo, autor, e.codigo,nombre  
from libros as l  
join editoriales as e  
on codigoeditorial=e.codigo;
```

-- Si eliminamos una editorial, se borra tal editorial de "editoriales" y todos los

-- registros de "libros" de dicha editorial:

```
delete from editoriales where codigo=2;
```

-- Veamos si el borrado se extendió a "libros":

```
select titulo, autor, e.codigo,nombre  
from libros as l  
join editoriales as e  
on codigoeditorial=e.codigo;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL 11  
45 -- Si eliminamos una editorial, se borra tal editorial de "editoriales" y todos los  
46 -- registros de "libros" de dicha editorial:  
47 delete from editoriales where codigo=2;  
48  
49 -- Veamos si el borrado se extendió a "libros":  
50 select titulo, autor, e.codigo,nombre  
51 from libros as l  
52 join editoriales as e  
53 on codigoeditorial=e.codigo;
```

Data Output	Explain	Messages	Notifications	Query History												
<table border="1"><thead><tr><th>titulo</th><th>autor</th><th>codigo</th><th>nombre</th></tr><tr><th>character varying (40)</th><th>character varying (30)</th><th>integer</th><th>character varying (20)</th></tr></thead><tbody><tr><td>El aleph</td><td>Borges</td><td>10</td><td>Emece</td></tr></tbody></table>	titulo	autor	codigo	nombre	character varying (40)	character varying (30)	integer	character varying (20)	El aleph	Borges	10	Emece				
titulo	autor	codigo	nombre													
character varying (40)	character varying (30)	integer	character varying (20)													
El aleph	Borges	10	Emece													

57 - Unión

El operador "union" combina el resultado de dos o más instrucciones "select" en un único resultado.

Se usa cuando los datos que se quieren obtener pertenecen a distintas tablas y no se puede acceder a ellos con una sola consulta.

Es necesario que las tablas referenciadas tengan tipos de datos similares, la misma cantidad de campos y el mismo orden de campos en la lista de selección de cada consulta. No se incluyen las filas duplicadas en el resultado, a menos que coloque la opción "all".

Se deben especificar los nombres de los campos en la primera instrucción "select".

Puede emplear la cláusula "order by".

Puede dividir una consulta compleja en varias consultas "select" y luego emplear el operador "union" para combinarlas.

Una academia de enseñanza almacena los datos de los alumnos en una tabla llamada "alumnos" y los datos de los profesores en otra denominada "profesores".

La academia necesita el nombre y domicilio de profesores y alumnos para enviarles una tarjeta de invitación.

Para obtener los datos necesarios de ambas tablas en una sola consulta necesitamos realizar una unión:

```
select nombre, domicilio from alumnos  
union  
select nombre, domicilio from profesores;
```

El primer "select" devuelve el nombre y domicilio de todos los alumnos; el segundo, el nombre y domicilio de todos los profesores.

Los encabezados del resultado de una unión son los que se especifican en el primer "select".

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists profesores;  
drop table if exists alumnos;
```

```
create table profesores(  
documento varchar(8) not null,  
nombre varchar (30),
```

```

domicilio varchar(30),
primary key(documento)

);

create table alumnos(
documento varchar(8) not null,
nombre varchar (30),
domicilio varchar(30),
primary key(documento)

);

insert into alumnos values('30000000','Juan Perez','Colon 123');

insert into alumnos values('30111111','Marta Morales','Caseros 222');

insert into alumnos values('30222222','Laura Torres','San Martin 987');

insert into alumnos values('30333333','Mariano Juarez','Avellaneda 34');

insert into alumnos values('23333333','Federico Lopez','Colon 987');

insert into profesores values('22222222','Susana Molina','Sucre 345');

insert into profesores values('23333333','Federico Lopez','Colon 987');

```

-- La academia necesita el nombre y domicilio de profesores y alumnos para
-- enviarles una tarjeta de invitación.

-- Empleamos el operador "union" para obtener dicha información de ambas tablas:

```

select nombre, domicilio from alumnos
union
select nombre, domicilio from profesores;

```

-- Note que existe un profesor que también está presente en la tabla "alumnos";

-- dicho registro aparece una sola vez en el resultado de "union".

-- Si queremos que las filas duplicadas aparezcan, debemos emplear "all":

```
select nombre, domicilio from alumnos
```

```
union all
```

```
select nombre, domicilio from profesores;
```

-- Ordenamos por domicilio:

```
select nombre, domicilio from alumnos
```

```
union
```

```
select nombre, domicilio from profesores
```

```
order by domicilio;
```

-- Podemos agregar una columna extra a la consulta con el encabezado

-- "condicion" en la que aparezca el literal "profesor" o "alumno"

-- según si la persona es uno u otro:

```
select nombre, domicilio, 'alumno' as condicion from alumnos
```

```
union
```

```
select nombre, domicilio,'profesor' from profesores
```

```
order by condicion;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL-11
44
45 -- Podemos agregar una columna extra a la consulta con el encabezado
46 -- "condición" en la que aparezca el literal "profesor" o "alumno"
47 -- según si la persona es uno u otro:
48 select nombre, domicilio, 'alumno' as condicion from alumnos
49 union
50   select nombre, domicilio,'profesor' from profesores
51 order by condicion;
52
```

	nombre	domicilio	condicion
1	Mariano Juarez	Avellaneda 34	alumno
2	Marta Morales	Caseros 222	alumno
3	Laura Torres	San Martin 987	alumno
4	Juan Perez	Colon 123	alumno
5	Federico Lopez	Colon 987	alumno
6	Susana Molina	Sucre 345	profesor
7	Federico Lopez	Colon 987	profesor

58 - Subconsultas

Una subconsulta (subquery) es una sentencia "select" anidada en otra sentencia "select", "insert", "update" o "delete" (o en otra subconsulta).

Las subconsultas se emplean cuando una consulta es muy compleja, entonces se la divide en varios pasos lógicos y se obtiene el resultado con una única instrucción y cuando la consulta depende de los resultados de otra consulta.

Generalmente, una subconsulta se puede reemplazar por combinaciones y estas últimas son más eficientes.

Las subconsultas se DEBEN incluir entre paréntesis.

Puede haber subconsultas dentro de subconsultas.

Se pueden emplear subconsultas:

- en lugar de una expresión, siempre que devuelvan un solo valor o una lista de valores.
- que retornen un conjunto de registros de varios campos en lugar de una tabla o para obtener el mismo resultado que una combinación (join).

Hay tres tipos básicos de subconsultas:

las que retornan un solo valor escalar que se utiliza con un operador de comparación o en lugar de una expresión.

las que retornan una lista de valores, se combinan con "in", o los operadores "any", "some" y "all".

los que testean la existencia con "exists".

Reglas a tener en cuenta al emplear subconsultas:

- la lista de selección de una subconsulta que va luego de un operador de comparación puede incluir sólo una expresión o campo (excepto si se emplea "exists" y "in").
- si el "where" de la consulta exterior incluye un campo, este debe ser compatible con el campo en la lista de selección de la subconsulta.
- las subconsultas luego de un operador de comparación (que no es seguido por "any" o "all") no pueden incluir cláusulas "group by" ni "having".
- "distinct" no puede usarse con subconsultas que incluyan "group by".
- una subconsulta puede estar anidada dentro del "where" o "having" de una consulta externa o dentro de otra subconsulta.
- si una tabla se nombra solamente en un subconsulta y no en la consulta externa, los campos no serán incluidos en la salida (en la lista de selección de la consulta externa).

59 - Subconsultas como expresión

Una subconsulta puede reemplazar una expresión. Dicha subconsulta debe devolver un valor escalar (o una lista de valores de un campo).

Las subconsultas que retornan un solo valor escalar se utiliza con un operador de comparación o en lugar de una expresión:

```
select CAMPOS  
from TABLA  
where CAMPO OPERADOR (SUBCONSULTA);
```

```
select CAMPO OPERADOR (SUBCONSULTA)  
from TABLA;
```

Si queremos saber el precio de un determinado libro y la diferencia con el precio del libro más costoso, anteriormente debíamos averiguar en una consulta el precio del libro más costoso y luego, en otra consulta, calcular la diferencia con el valor del libro que solicitamos. Podemos conseguirlo en una sola sentencia combinando dos consultas:

```
select titulo,precio,  
precio-(select max(precio) from libros) as diferencia  
from libros  
where titulo='Uno';
```

En el ejemplo anterior se muestra el título, el precio de un libro y la diferencia entre el precio del libro y el máximo valor de precio.

Queremos saber el título, autor y precio del libro más costoso:

```
select titulo,autor, precio  
from libros  
where precio=  
(select max(precio) from libros);
```

Note que el campo del "where" de la consulta exterior es compatible con el valor retornado por la expresión de la subconsulta.

Se pueden emplear en "select", "insert", "update" y "delete".

Para actualizar un registro empleando subconsulta la sintaxis básica es la siguiente:

```
update TABLA set CAMPO=NUEVOVALOR
```

```
where CAMPO= (SUBCONSULTA);
```

Para eliminar registros empleando subconsulta empleamos la siguiente sintaxis básica:

```
delete from TABLA
```

```
where CAMPO=(SUBCONSULTA);
```

Recuerde que la lista de selección de una subconsulta que va luego de un operador de comparación puede incluir sólo una expresión o campo (excepto si se emplea "exists" o "in").

No olvide que las subconsultas luego de un operador de comparación (que no es seguido por "any" o "all") no pueden incluir cláusulas "group by".

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
codigo serial,
```

```
titulo varchar(40),
```

```
autor varchar(30),
```

```
editorial varchar(20),
```

```
precio decimal(5,2),
```

```
primary key(codigo)
```

```
);
```

```
insert into libros(titulo,autor,editorial,precio)
```

```
values('Alicia en el pais de las maravillas','Lewis Carroll','Emece',20.00);
```

```
insert into libros(titulo,autor,editorial,precio)
```

```
values('Alicia en el pais de las maravillas','Lewis Carroll','Plaza',35.00);
```

```
insert into libros(titulo,autor,editorial,precio)
```

```
values('Aprenda PHP','Mario Molina','Siglo XXI',40.00);
```

```
insert into libros(titulo,autor,editorial,precio)
```

```
values('El aleph','Borges','Emece',10.00);
```

```
insert into libros(titulo,autor,editorial,precio)
values('Ilusiones','Richard Bach','Planeta',15.00);

insert into libros(titulo,autor,editorial,precio)
values('Java en 10 minutos','Mario Molina','Siglo XXI',50.00);

insert into libros(titulo,autor,editorial,precio)
values('Martin Fierro','Jose Hernandez','Planeta',20.00);

insert into libros(titulo,autor,editorial,precio)
values('Martin Fierro','Jose Hernandez','Emece',30.00);

insert into libros(titulo,autor,editorial,precio)
values('Uno','Richard Bach','Planeta',10.00);
```

-- Obtenemos el título, precio de un libro específico y

-- la diferencia entre su precio y el máximo valor:

```
select titulo,precio,
precio-(select max(precio) from libros) as diferencia
from libros
where titulo='Uno';
```

-- Mostramos el título y precio del libro más costoso:

```
select titulo,autor, precio
from libros
where precio=
(select max(precio) from libros);
```

-- Actualizamos el precio del libro con máximo valor:

```
update libros set precio=45
where precio=
(select max(precio) from libros);
```

-- Eliminamos los libros con precio menor:

```
delete from libros
```

```
where precio=
```

```
(select min(precio) from libros);
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL terminal window with the following content:

```
bd1 on postgres@PostgreSQL11
values('Uno','Richard Bach','Planeta',10.00);
-- Obtenemos el título, precio de un libro específico y
-- la diferencia entre su precio y el máximo valor:
select titulo,precio,
precio-(select max(precio) from libros) as diferencia
from libros
where titulo='Uno';
```

Below the code, there is a table output:

	titulo character varying (40)	precio numeric (5,2)	diferencia numeric
1	Uno	10.00	-40.00

60 - Subconsultas con in

Vimos que una subconsulta puede reemplazar una expresión. Dicha subconsulta debe devolver un valor escalar o una lista de valores de un campo; las subconsultas que retornan una lista de valores reemplazan a una expresión en una cláusula "where" que contiene la palabra clave "in".

El resultado de una subconsulta con "in" (o "not in") es una lista. Luego que la subconsulta retorna resultados, la consulta exterior los usa.

La sintaxis básica es la siguiente:

```
...where EXPRESION in (SUBCONSULTA);
```

Este ejemplo muestra los nombres de las editoriales que han publicado libros de un determinado autor:

```
select nombre  
from editoriales  
where codigo in  
(select codigoeditorial  
from libros  
where autor='Richard Bach');
```

La subconsulta (consulta interna) retorna una lista de valores de un solo campo (codigo) que la consulta exterior luego emplea al recuperar los datos.

Podemos reemplazar por un "join" la consulta anterior:

```
select distinct nombre  
from editoriales as e  
join libros  
on codigoeditorial=e.codigo  
where autor='Richard Bach';
```

Una combinación (join) siempre puede ser expresada como una subconsulta; pero una subconsulta no siempre puede reemplazarse por una combinación que retorne el mismo resultado. Si es posible, es aconsejable emplear combinaciones en lugar de subconsultas, son más eficientes.

Se recomienda probar las subconsultas antes de incluirlas en una consulta exterior, así puede verificar que retorna lo necesario, porque a veces resulta difícil verlo en consultas anidadas.

También podemos buscar valores No coincidentes con una lista de valores que retorna una subconsulta; por ejemplo, las editoriales que no han publicado libros de un autor específico:

```
select nombre  
from editoriales  
where codigo not in  
(select codigoeditorial  
from libros  
where autor='Richard Bach');
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists editoriales;  
drop table if exists libros;
```

```
create table editoriales(  
codigo serial,  
nombre varchar(30),  
primary key (codigo)  
);
```

```
create table libros (  
codigo serial,  
titulo varchar(40),  
autor varchar(30),  
codigoeditorial smallint,  
primary key(codigo)  
);
```

```
insert into editoriales(nombre) values('Planeta');  
insert into editoriales(nombre) values('Emece');  
insert into editoriales(nombre) values('Paidos');
```

```
insert into editoriales(nombre) values('Siglo XXI');

insert into libros(titulo,autor,codigoeditorial) values('Uno','Richard Bach',1);
insert into libros(titulo,autor,codigoeditorial) values('Ilusiones','Richard Bach',1);
insert into libros(titulo,autor,codigoeditorial) values('Aprenda PHP','Mario Molina',4);
insert into libros(titulo,autor,codigoeditorial) values('El aleph','Borges',2);
insert into libros(titulo,autor,codigoeditorial) values('Puente al infinito','Richard Bach',2);

-- Queremos conocer el nombre de las editoriales que han publicado libros del autor "Richard Bach":
select nombre
from editoriales
where codigo in
(select codigoeditorial
from libros
where autor='Richard Bach');

-- Probamos la subconsulta separada de la consulta exterior para verificar que retorna una lista
-- de valores de un solo campo:
select codigoeditorial
from libros
where autor='Richard Bach';

-- Podemos reemplazar por un "join" la primera consulta:
select distinct nombre
from editoriales as e
join libros
on codigoeditorial=e.codigo
where autor='Richard Bach';
```

-- También podemos buscar las editoriales que no han publicado libros de "Richard Bach":

```
select nombre
from editoriales
where codigo not in
(select codigoeditorial
from libros
where autor='Richard Bach');
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL terminal window with the following content:

```
bd1 on postgres@PostgreSQL:11
21 insert into editoriales(nombre) values('Siglo XXI');
22
23 insert into libros(titulo,autor,codigoeditorial) values('Uno','Richard Bach',1);
24 insert into libros(titulo,autor,codigoeditorial) values('Ilusiones','Richard Bach',1);
25 insert into libros(titulo,autor,codigoeditorial) values('Aprenda PHP','Mario Molina',4);
26 insert into libros(titulo,autor,codigoeditorial) values('El aleph','Borges',2);
27 insert into libros(titulo,autor,codigoeditorial) values('Puente al infinito','Richard Bach',2);
28
29 -- Queremos conocer el nombre de las editoriales que han publicado libros del autor "Richard Bach":
30 select nombre
31   from editoriales
32  where codigo in
33    (select codigoeditorial
34      from libros
35     where autor='Richard Bach');
36
```

The output section shows a table with the following data:

nombre
Planeta
Emece

61 - Subconsultas any - some - all

"any" y "some" son sinónimos. Chequean si alguna fila de la lista resultado de una subconsulta se encuentra el valor especificado en la condición.

Compara un valor escalar con los valores de un campo y devuelven "true" si la comparación con cada valor de la lista de la subconsulta es verdadera, sino "false".

El tipo de datos que se comparan deben ser compatibles.

La sintaxis básica es:

...VALORESCALAR OPERADORDECOMPARACION

ANY (SUBCONSULTA);

Queremos saber los títulos de los libros de "Borges" que pertenecen a editoriales que han publicado también libros de "Richard Bach", es decir, si los libros de "Borges" coinciden con ALGUNA de las editoriales que publicó libros de "Richard Bach":

```
select titulo
from libros
where autor='Borges' and
codigoeditorial = any
(select e.codigo
from editoriales as e
join libros as l
on codigoeditorial=e.codigo
where l.autor='Richard Bach');
```

La consulta interna (subconsulta) retorna una lista de valores de un solo campo (puede ejecutar la subconsulta como una consulta para probarla), luego, la consulta externa compara cada valor de "codigoeditorial" con cada valor de la lista devolviendo los títulos de "Borges" que coinciden.

"all" también compara un valor escalar con una serie de valores. Chequea si TODOS los valores de la lista de la consulta externa se encuentran en la lista de valores devuelta por la consulta interna. Sintaxis:

VALORESCALAR OPERADORDECOMPARACION all (SUBCONSULTA);

Queremos saber si TODAS las editoriales que publicaron libros de "Borges" coinciden con TODAS las editoriales que publicaron libros de "Richard Bach":

```
select titulo
```

```
from libros

where autor='Borges' and

codigoeditorial = all

(select e.codigo

from editoriales as e

join libros as l

on codigoeditorial=e.codigo

where l.autor='Richard Bach');
```

La consulta interna (subconsulta) retorna una lista de valores de un solo campo (puede ejecutar la subconsulta como una consulta para probarla), luego, la consulta externa compara cada valor de "codigoeditorial" con cada valor de la lista, si TODOS coinciden, devuelve los títulos.

Veamos otro ejemplo con un operador de comparación diferente:

Queremos saber si ALGUN precio de los libros de "Borges" es mayor a ALGUN precio de los libros de "Richard Bach":

```
select titulo,precio

from libros

where autor='Borges' and

precio > any

(select precio

from libros

where autor='Bach');
```

El precio de cada libro de "Borges" es comparado con cada valor de la lista de valores retornada por la subconsulta; si ALGUNO cumple la condición, es decir, es mayor a ALGUN precio de "Richard Bach", se lista.

Veamos la diferencia si empleamos "all" en lugar de "any":

```
select titulo,precio

from libros

where autor='borges' and

precio > all

(select precio

from libros
```

```
where autor='bach');
```

El precio de cada libro de "Borges" es comparado con cada valor de la lista de valores retornada por la subconsulta; si cumple la condición, es decir, si es mayor a TODOS los precios de "Richard Bach" (o al mayor), se lista.

Emplear "= any" es lo mismo que emplear "in".

Emplear "<> all" es lo mismo que emplear "not in".

Recuerde que solamente las subconsultas luego de un operador de comparación al cual es seguido por "any" o "all") pueden incluir cláusulas "group by".

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists editoriales;
```

```
drop table if exists libros;
```

```
create table editoriales(
```

```
codigo serial,
```

```
nombre varchar(30),
```

```
primary key (codigo)
```

```
);
```

```
create table libros (
```

```
codigo serial,
```

```
titulo varchar(40),
```

```
autor varchar(30),
```

```
codigoeeditorial smallint,
```

```
precio decimal(5,2),
```

```
primary key(codigo)
```

```
);
```

```
insert into editoriales(nombre) values('Planeta');
```

```
insert into editoriales(nombre) values('Emece');
```

```
insert into editoriales(nombre) values('Paidos');
```

```
insert into editoriales(nombre) values('Siglo XXI');

insert into libros(titulo,autor,codigoeditorial,precio) values('Uno','Richard Bach',1,15);
insert into libros(titulo,autor,codigoeditorial,precio) values('Ilusiones','Richard Bach',4,18);
insert into libros(titulo,autor,codigoeditorial,precio) values('Puente al infinito','Richard Bach',2,20);
insert into libros(titulo,autor,codigoeditorial,precio) values('Aprenda PHP','Mario Molina',4,40);
insert into libros(titulo,autor,codigoeditorial,precio) values('El aleph','Borges',2,10);
insert into libros(titulo,autor,codigoeditorial,precio) values('Antología','Borges',1,20);
insert into libros(titulo,autor,codigoeditorial,precio) values('Cervantes y el Quijote','Borges',3,25);
```

-- Mostramos los títulos de los libros de "Borges" de editoriales que han publicado

-- también libros de "Richard Bach":

```
select titulo
from libros
where autor like '%Borges%' and
      codigoeditorial = any
      (select e.codigo
       from editoriales as e
       join libros as l
       on codigoeditorial=e.codigo
       where l.autor like '%Bach%');
```

-- Realizamos la misma consulta pero empleando "all" en lugar de "any":

```
select titulo
from libros
where autor like '%Borges%' and
      codigoeditorial = all
      (select e.codigo
```

```
from editoriales as e  
join libros as l  
on codigoeditorial=e.codigo  
where l.autor like '%Bach%');
```

-- Mostramos los títulos y precios de los libros "Borges" cuyo precio supera

-- a ALGUN precio de los libros de "Richard Bach":

```
select titulo,precio  
from libros  
where autor like '%Borges%' and  
precio > any  
(select precio  
from libros  
where autor like '%Bach%');
```

-- Veamos la diferencia si empleamos "all" en lugar de "any":

```
select titulo,precio  
from libros  
where autor like '%Borges%' and  
precio > all  
(select precio  
from libros  
where autor like '%Bach%');
```

-- Empleamos la misma subconsulta para eliminación:

```
delete from libros  
where autor like '%Borges%' and  
precio > all  
(select precio
```

```
from libros  
where autor like '%Bach%');
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL query editor interface. The top part displays a code block with numbered lines. Lines 32 through 42 are highlighted with a red box, containing a complex multi-table query. The bottom part shows the execution results in a table format.

	titulo
1	El aleph
2	Antología

62 - Subconsultas correlacionadas

Un almacén almacena la información de sus ventas en una tabla llamada "facturas" en la cual guarda el número de factura, la fecha y el nombre del cliente y una tabla denominada "detalles" en la cual se almacenan los distintos items correspondientes a cada factura: el nombre del artículo, el precio (unitario) y la cantidad.

Se necesita una lista de todas las facturas que incluya el número, la fecha, el cliente, la cantidad de artículos comprados y el total:

```
select f.*,
       (select count(d.numeroitem)
        from Detalles as d
        where f.numero=d.numerofactura) as cantidad,
       (select sum(d.preciounitario*cantidad)
        from Detalles as d
        where f.numero=d.numerofactura) as total
       from facturas as f;
```

El segundo "select" retorna una lista de valores de una sola columna con la cantidad de items por factura (el número de factura lo toma del "select" exterior); el tercer "select" retorna una lista de valores de una sola columna con el total por factura (el número de factura lo toma del "select" exterior); el primer "select" (externo) devuelve todos los datos de cada factura.

A este tipo de subconsulta se la denomina consulta correlacionada. La consulta interna se evalúa tantas veces como registros tiene la consulta externa, se realiza la subconsulta para cada registro de la consulta externa. El campo de la tabla dentro de la subconsulta (f.numero) se compara con el campo de la tabla externa.

En este caso, específicamente, la consulta externa pasa un valor de "numero" a la consulta interna. La consulta interna toma ese valor y determina si existe en "detalles", si existe, la consulta interna devuelve la suma. El proceso se repite para el registro de la consulta externa, la consulta externa pasa otro "numero" a la consulta interna y PostgreSQL repite la evaluación.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists facturas;
```

```
drop table if exists detalles;
```

```
create table facturas(
```

```
    numero int not null,
```

```

fecha date,
cliente varchar(30),
primary key(numero)
);

create table detalles(
numerofactura int not null,
numeroitem int not null,
articulo varchar(30),
precio decimal(5,2),
cantidad int,
primary key(numerofactura,numeroitem)
);

insert into facturas values(1200,'2018-01-15','Juan Lopez');
insert into facturas values(1201,'2018-01-15','Luis Torres');
insert into facturas values(1202,'2018-01-15','Ana Garcia');
insert into facturas values(1300,'2018-01-20','Juan Lopez');

insert into detalles values(1200,1,'lapiz',1,100);
insert into detalles values(1200,2,'goma',0.5,150);
insert into detalles values(1201,1,'regla',1.5,80);
insert into detalles values(1201,2,'goma',0.5,200);
insert into detalles values(1201,3,'cuaderno',4,90);
insert into detalles values(1202,1,'lapiz',1,200);
insert into detalles values(1202,2,'escuadra',2,100);
insert into detalles values(1300,1,'lapiz',1,300);

-- Se necesita una lista de todas las facturas que incluya el número,

```

-- la fecha, el cliente, la cantidad de artículos comprados y el total:

```
select f.*,
       (select count(d.numeroitem)
        from detalles as d
        where f.numero=d.numerofactura) as cantidad,
       (select sum(d.precio*cantidad)
        from detalles as d
        where f.numero=d.numerofactura) as total
     from facturas as f;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL terminal window titled "bd1 on postgres@PostgreSQL:11". The command history (lines 19-44) contains SQL statements for inserting data into the "facturas" and "detalles" tables, and a complex query (lines 34-44) that selects the number of items, quantity, and total price for each invoice.

```
19
20 insert into facturas values(1200,'2018-01-15','Juan Lopez');
21 insert into facturas values(1201,'2018-01-15','Luis Torres');
22 insert into facturas values(1202,'2018-01-15','Ana Garcia');
23 insert into facturas values(1300,'2018-01-20','Juan Lopez');

24
25 insert into detalles values(1200,1,'lapiz',1,100);
26 insert into detalles values(1200,2,'goma',0.5,150);
27 insert into detalles values(1201,1,'regla',1.5,80);
28 insert into detalles values(1201,2,'goma',0.5,200);
29 insert into detalles values(1201,3,'cuaderno',4,90);
30 insert into detalles values(1202,1,'lapiz',1,200);
31 insert into detalles values(1202,2,'escuadra',2,100);
32 insert into detalles values(1300,1,'lapiz',1,300);

33
34 -- Se necesita una lista de todas las facturas que incluya el número,
35 -- la fecha, el cliente, la cantidad de artículos comprados y el total:
36 select f.*,
37       (select count(d.numeroitem)
38        from detalles as d
39        where f.numero=d.numerofactura) as cantidad,
40       (select sum(d.precio*cantidad)
41        from detalles as d
42        where f.numero=d.numerofactura) as total
43     from facturas as f;
44
```

The bottom part of the screenshot shows the "Data Output" tab selected, displaying a table with four rows of data corresponding to the invoices inserted earlier. The columns are: numero (integer), fecha (date), cliente (character varying(30)), cantidad (bigint), and total (numeric).

	numero	fecha	cliente	cantidad	total
	integer	date	character varying(30)	bigint	numeric
1	1200	2018-01-15	Juan Lopez	2	175.00
2	1201	2018-01-15	Luis Torres	3	580.00
3	1202	2018-01-15	Ana Garcia	2	400.00
4	1300	2018-01-20	Juan Lopez	1	300.00

63 - Subconsultas (Exists y Not Exists)

Los operadores "exists" y "not exists" se emplean para determinar si hay o no datos en una lista de valores.

Estos operadores pueden emplearse con subconsultas correlacionadas para restringir el resultado de una consulta exterior a los registros que cumplen la subconsulta (consulta interior). Estos operadores retornan "true" (si las subconsultas retornan registros) o "false" (si las subconsultas no retornan registros).

Cuando se coloca en una subconsulta el operador "exists", PostgreSQL analiza si hay datos que coinciden con la subconsulta, no se devuelve ningún registro, es como un test de existencia; PostgreSQL termina la recuperación de registros cuando por lo menos un registro cumple la condición "where" de la subconsulta.

La sintaxis básica es la siguiente:

```
... where exists (SUBCONSULTA);
```

En este ejemplo se usa una subconsulta correlacionada con un operador "exists" en la cláusula "where" para devolver una lista de clientes que compraron el artículo "lapiz":

```
select cliente,numero  
from facturas as f  
where exists  
(select *from Detalles as d  
where f.numero=d.numerofactura  
and d.articulo='lapiz');
```

Puede obtener el mismo resultado empleando una combinación.

Podemos buscar los clientes que no han adquirido el artículo "lapiz" empleando "if not exists":

```
select cliente,numero  
from facturas as f  
where not exists  
(select *from Detalles as d  
where f.numero=d.numerofactura  
and d.articulo='lapiz');
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists facturas;
```

```
drop table if exists detalles;
```

```
create table facturas(
```

```
numero int not null,
```

```
fecha date,
```

```
cliente varchar(30),
```

```
primary key(numero)
```

```
);
```

```
create table detalles(
```

```
numerofactura int not null,
```

```
numeroitem int not null,
```

```
articulo varchar(30),
```

```
precio decimal(5,2),
```

```
cantidad int,
```

```
primary key(numerofactura,numeroitem)
```

```
);
```

```
insert into facturas values(1200,'2018-01-15','Juan Lopez');
```

```
insert into facturas values(1201,'2018-01-15','Luis Torres');
```

```
insert into facturas values(1202,'2018-01-15','Ana Garcia');
```

```
insert into facturas values(1300,'2018-01-20','Juan Lopez');
```

```
insert into detalles values(1200,1,'lapiz',1,100);
```

```
insert into detalles values(1200,2,'goma',0.5,150);
```

```
insert into detalles values(1201,1,'regla',1.5,80);
```

```
insert into detalles values(1201,2,'goma',0.5,200);
```

```

insert into detalles values(1201,3,'cuaderno',4,90);

insert into detalles values(1202,1,'lapiz',1,200);

insert into detalles values(1202,2,'escuadra',2,100);

insert into detalles values(1300,1,'lapiz',1,300);

-- Empleamos una subconsulta correlacionada con un operador "exists"

-- en la cláusula "where" para devolver la lista de clientes que

-- compraron el artículo "lapiz":

select cliente,numero

from facturas as f

where exists

(select *from detalles as d

where f.numero=d.numerofactura

and d.articulo='lapiz');

-- Buscamos los clientes que NO han comprado el artículo "lapiz":

select cliente,numero

from facturas as f

where not exists

(select *from detalles as d

where f.numero=d.numerofactura

and d.articulo='lapiz');

```

La ejecución de este lote de comandos SQL genera una salida similar a:

```

bd1 on postgres@PostgreSQL 11
26 insert into detalles values(1200,2,'goma',0.5,150);
27 insert into detalles values(1201,1,'regla',1.5,80);
28 insert into detalles values(1201,2,'goma',0.5,200);
29 insert into detalles values(1201,3,'cuaderno',4,90);
30 insert into detalles values(1202,1,'lapiz',1,200);
31 insert into detalles values(1202,2,'escuadra',2,100);
32 insert into detalles values(1300,1,'lapiz',1,300);
33
34 -- Empleamos una subconsulta correlacionada con un operador "exists"
35 -- en la cláusula "where" para devolver la lista de clientes que
36 -- compraron el artículo "lapiz":
37 select cliente,numero
38   from facturas as f
39 where exists
40   (select *from detalles as d
41     where f.numero=d.numeroefactura
42     and d.articulo='lapiz');
43
44 -- Buscamos los clientes que NO han comprado el artículo "lapiz":
45 select cliente,numero
46   from facturas as f
47 where not exists
48   (select *from detalles as d
49     where f.numero=d.numeroefactura
50     and d.articulo='lapiz');
51

```

Data Output Explain Messages Notifications Query History

	cliente character varying (30)	numero integer
1	Luis Torres	1201

64 - Subconsulta similar autocombinación

Algunas sentencias en las cuales la consulta interna y la externa emplean la misma tabla pueden reemplazarse por una autocombinación.

Por ejemplo, queremos una lista de los libros que han sido publicados por distintas editoriales.

```
select distinct l1.titulo  
from libros as l1  
where l1.titulo in  
(select l2.titulo  
from libros as l2  
where l1.editorial <> l2.editorial);
```

En el ejemplo anterior empleamos una subconsulta correlacionada y las consultas interna y externa emplean la misma tabla. La subconsulta devuelve una lista de valores por ello se emplea "in" y sustituye una expresión en una cláusula "where".

Con el siguiente "join" se obtiene el mismo resultado:

```
select distinct l1.titulo  
from libros as l1  
join libros as l2  
on l1.titulo=l1.titulo and  
l1.autor=l2.autor  
where l1.editorial<>l2.editorial;
```

Otro ejemplo: Buscamos todos los libros que tienen el mismo precio que "El aleph" empleando subconsulta:

```
select titulo  
from libros  
where titulo<>'El aleph' and  
precio =  
(select precio  
from libros  
where titulo='El aleph');
```

La subconsulta retorna un solo valor.

Buscamos los libros cuyo precio supere el precio promedio de los libros por editorial:

```
select l1.titulo,l1.editorial,l1.precio  
from libros as l1  
where l1.precio >  
(select avg(l2.precio)  
from libros as l2  
where l1.editorial= l2.editorial);
```

Por cada valor de l1, se evalúa la subconsulta, si el precio es mayor que el promedio.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
create table libros(
```

```
codigo serial,
```

```
titulo varchar(40),
```

```
autor varchar(30),
```

```
editorial varchar(20),
```

```
precio decimal(5,2),
```

```
primary key(codigo)
```

```
);
```

```
insert into libros(titulo,autor,editorial,precio)
```

```
values('Alicia en el pais de las maravillas','Lewis Carroll','Emece',20.00);
```

```
insert into libros(titulo,autor,editorial,precio)
```

```
values('Alicia en el pais de las maravillas','Lewis Carroll','Plaza',35.00);
```

```
insert into libros(titulo,autor,editorial,precio)
```

```
values('Aprenda PHP','Mario Molina','Siglo XXI',40.00);
```

```
insert into libros(titulo,autor,editorial,precio)
```

```
values('El aleph','Borges','Emece',10.00);
```

```
insert into libros(titulo,autor,editorial,precio)
values('Ilusiones','Richard Bach','Planeta',15.00);

insert into libros(titulo,autor,editorial,precio)
values('Java en 10 minutos','Mario Molina','Siglo XXI',50.00);

insert into libros(titulo,autor,editorial,precio)
values('Martin Fierro','Jose Hernandez','Planeta',20.00);

insert into libros(titulo,autor,editorial,precio)
values('Martin Fierro','Jose Hernandez','Emece',30.00);

insert into libros(titulo,autor,editorial,precio)
values('Uno','Richard Bach','Planeta',10.00);
```

-- Obtenemos la lista de los libros que han sido publicados por distintas

-- editoriales empleando una consulta correlacionada:

```
select distinct l1.titulo
from libros as l1
where l1.titulo in
(select l2.titulo
from libros as l2
where l1.editorial <> l2.editorial);
```

-- El siguiente "join" retorna el mismo resultado:

```
select distinct l1.titulo
from libros as l1
join libros as l2
on l1.titulo=l2.titulo
where l1.editorial<>l2.editorial;
```

-- Buscamos todos los libros que tienen el mismo precio que

-- "El aleph" empleando subconsulta:

```
select titulo  
from libros  
where titulo<>'El aleph' and  
precio =  
(select precio  
from libros  
where titulo='El aleph');
```

-- Obtenemos la misma salida empleando "join":

```
select l1.titulo  
from libros as l1  
join libros as l2  
on l1.precio=l2.precio  
where l2.titulo='El aleph' and  
l1.titulo<>l2.titulo;
```

-- Buscamos los libros cuyo precio supera el precio promedio

-- de los libros por editorial:

```
select l1.titulo,l1.editorial,l1.precio  
from libros as l1  
where l1.precio >  
(select avg(l2.precio)  
from libros as l2  
where l1.editorial= l2.editorial);
```

-- Obtenemos la misma salida pero empleando un "join" con "having":

```
select l1.editorial,l1.titulo,l1.precio  
from libros as l1  
join libros as l2
```

```

on l1.editorial=l2.editorial
group by l1.editorial, l1.titulo, l1.precio
having l1.precio > avg(l2.precio);

```

La ejecución de este lote de comandos SQL genera una salida similar a:

```

bd1 on postgres@PostgreSQL 11
22  Insert into libros(titulo,autor,editorial,precio)
23    values('Java en 10 minutos','Mario Molina','Siglo XXI',50.00);
24  insert into libros(titulo,autor,editorial,precio)
25    values('Martin Fierro','Jose Hernandez','Planeta',20.00);
26  insert into libros(titulo,autor,editorial,precio)
27    values('Martin Fierro','Jose Hernandez','Emece',30.00);
28  insert into libros(titulo,autor,editorial,precio)
29    values('Uno','Richard Bach','Planeta',10.00);
30
31  -- Obtenemos la lista de los libros que han sido publicados por distintas
32  -- editoriales empleando una consulta correlacionada:
33  select distinct ll.titulo
34    from libros as ll
35   where ll.titulo in
36     (select l2.titulo
37      from libros as l2
38     where ll.editorial <> l2.editorial);
39

```

Data Output Explain Messages Notifications Query History

	titulo
▲	character varying (40)
1	Alicia en el pais de las ma...
2	Martin Fierro

65 - Subconsulta en lugar de una tabla

Se pueden emplear subconsultas que retornen un conjunto de registros de varios campos en lugar de una tabla.

Se la denomina tabla derivada y se coloca en la cláusula "from" para que la use un "select" externo.

La tabla derivada debe ir entre paréntesis y tener un alias para poder referenciarla. La sintaxis básica es la siguiente:

```
select ALIASdeTABLADERIVADA.CAMPO
```

```
from (TABLADERIVADA) as ALIAS;
```

La tabla derivada es una subconsulta.

Podemos probar la consulta que retorna la tabla derivada y luego agregar el "select" externo:

```
select f.*,
```

```
(select sum(d.precio*cantidad)
```

```
from Detalles as d
```

```
where f.numero=d.numerofactura) as total
```

```
from facturas as f;
```

La consulta anterior contiene una subconsulta correlacionada; retorna todos los datos de "facturas" y el monto total por factura de "detalles". Esta consulta retorna varios registros y varios campos y será la tabla derivada que emplearemos en la siguiente consulta:

```
select td.numero,c.nombre,td.total
```

```
from clientes as c
```

```
join (select f.*,
```

```
(select sum(d.precio*cantidad)
```

```
from Detalles as d
```

```
where f.numero=d.numerofactura) as total
```

```
from facturas as f) as td
```

```
on td.codigocliente=c.codigo;
```

La consulta anterior retorna, de la tabla derivada (referenciada con "td") el número de factura y el monto total, y de la tabla "clientes", el nombre del cliente. Note que este "join" no emplea 2

tablas, sino una tabla propiamente dicha y una tabla derivada, que es en realidad una subconsulta.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists clientes;
```

```
drop table if exists facturas;
```

```
drop table if exists detalles;
```

```
create table clientes(
```

```
    codigo serial,
```

```
    nombre varchar(30),
```

```
    domicilio varchar(30),
```

```
    primary key(codigo)
```

```
);
```

```
create table facturas(
```

```
    numero int not null,
```

```
    fecha date,
```

```
    codigocliente int not null,
```

```
    primary key(numero)
```

```
);
```

```
create table detalles(
```

```
    numerofactura int not null,
```

```
    numeroitem int not null,
```

```
    articulo varchar(30),
```

```
    precio decimal(5,2),
```

```
    cantidad int,
```

```
    primary key(numerofactura,numeroitem)
```

```
);
```

```

insert into clientes(nombre,domicilio) values('Juan Lopez','Colon 123');

insert into clientes(nombre,domicilio) values('Luis Torres','Sucre 987');

insert into clientes(nombre,domicilio) values('Ana Garcia','Sarmiento 576');

insert into facturas values(1200,'2017-01-15',1);

insert into facturas values(1201,'2017-01-15',2);

insert into facturas values(1202,'2017-01-15',3);

insert into facturas values(1300,'2017-01-20',1);

insert into detalles values(1200,1,'lapiz',1,100);

insert into detalles values(1200,2,'goma',0.5,150);

insert into detalles values(1201,1,'regla',1.5,80);

insert into detalles values(1201,2,'goma',0.5,200);

insert into detalles values(1201,3,'cuaderno',4,90);

insert into detalles values(1202,1,'lapiz',1,200);

insert into detalles values(1202,2,'escuadra',2,100);

insert into detalles values(1300,1,'lapiz',1,300);

-- Vamos a realizar un "select" para recuperar el número de factura,
-- el código de cliente, la fecha y la suma total de todas las facturas:

select f.*,
       (select sum(d.precio*cantidad)
        from detalles as d
        where f.numero=d.numerofactura) as total
       from facturas as f;

-- Esta consulta contiene una subconsulta correlacionada.

-- Ahora utilizaremos el resultado de la consulta anterior como una tabla

```

```
-- derivada que emplearemos en lugar de una tabla para realizar un "join"
-- y recuperar el número de factura, el nombre del cliente y el monto total por factura:

select td.numero,c.nombre,td.total
from clientes as c
join (select f.*,
(select sum(d.precio*cantidad)
from detalles as d
where f.numero=d.numerofactura) as total
from facturas as f) as td
on td.codigocliente=c.codigo;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```

bd1 on postgres@PostgreSQL 11
46 -- Vamos a realizar un "select" para recuperar el número de factura,
47 -- el código de cliente, la fecha y la suma total de todas las facturas:
48 select f.*,
49   (select sum(d.precio*cantidad)
50    from detalles as d
51    where f.numero=d.numerofactura) as total
52  from facturas as f;
53 -- Esta consulta contiene una subconsulta correlacionada.
54
55 -- Ahora utilizaremos el resultado de la consulta anterior como una tabla
56 -- derivada que emplearemos en lugar de una tabla para realizar un "join".
57 -- y recuperar el número de factura, el nombre del cliente y el monto total por factura
58 select td.numero,c.nombre,td.total
59   from clientes as c
60 join (select f.*,
61   (select sum(d.precio*cantidad)
62    from detalles as d
63    where f.numero=d.numerofactura) as total
64   from facturas as f) as td
65   on td.codigocliente=c.codigo;
66

```

Data Output				Explain	Messages	Notifications	Query History
	numero integer	nombre character varying (30)	total numeric				
1	1200	Juan Lopez	175.00				
2	1201	Luis Torres	580.00				
3	1202	Ana Garcia	400.00				
4	1300	Juan Lopez	300.00				

66 - Subconsulta (update - delete)

Dijimos que podemos emplear subconsultas en sentencias "insert", "update", "delete", además de "select".

La sintaxis básica para realizar actualizaciones con subconsulta es la siguiente:

```
update TABLA set CAMPO=NUEVOVALOR
```

```
where CAMPO= (SUBCONSULTA);
```

Actualizamos el precio de todos los libros de editorial "Emece":

```
update libros set precio=precio+(precio*0.1)
```

```
where codigoeditorial=
```

```
(select codigo
```

```
from editoriales
```

```
where nombre='Emece');
```

La subconsulta retorna un único valor. También podemos hacerlo con un join.

La sintaxis básica para realizar eliminaciones con subconsulta es la siguiente:

```
delete from TABLA
```

```
where CAMPO in (SUBCONSULTA);
```

Eliminamos todos los libros de las editoriales que tiene publicados libros de "Juan Perez":

```
delete from libros
```

```
where codigoeditorial in
```

```
(select e.codigo
```

```
from editoriales as e
```

```
join libros
```

```
on codigoeditorial=e.codigo
```

```
where autor='Juan Perez');
```

La subconsulta es una combinación que retorna una lista de valores que la consulta externa emplea al seleccionar los registros para la eliminación.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists editoriales;
```

```
drop table if exists libros;
```

```
create table editoriales(  
    codigo serial,  
    nombre varchar(30),  
    primary key (codigo)  
);
```

```
create table libros (  
    codigo serial,  
    titulo varchar(40),  
    autor varchar(30),  
    codigoeditorial smallint,  
    precio decimal(5,2),  
    primary key(codigo)  
);
```

```
insert into editoriales(nombre) values('Planeta');  
insert into editoriales(nombre) values('Emece');  
insert into editoriales(nombre) values('Paidos');  
insert into editoriales(nombre) values('Siglo XXI');
```

```
insert into libros(titulo,autor,codigoeditorial,precio)  
values('Uno','Richard Bach',1,15);
```

```
insert into libros(titulo,autor,codigoeditorial,precio)  
values('Illusiones','Richard Bach',2,20);
```

```
insert into libros(titulo,autor,codigoeditorial,precio)  
values('El aleph','Borges',3,10);
```

```
insert into libros(titulo,autor,codigoeditorial,precio)  
values('Aprenda PHP','Mario Molina',4,40);
```

```

insert into libros(titulo,autor,codigoeditorial,precio)
values('Poemas','Juan Perez',1,20);

insert into libros(titulo,autor,codigoeditorial,precio)
values('Cuentos','Juan Perez',3,25);

insert into libros(titulo,autor,codigoeditorial,precio)
values('Java en 10 minutos','Marcelo Perez',2,30);

-- Actualizamos el precio de todos los libros de editorial "Emece" incrementándolos en un 10%:
update libros set precio=precio+(precio*0.1)

where codigoeditorial=
(select codigo
from editoriales
where nombre='Emece');

-- Eliminamos todos los libros de las editoriales que tiene publicados libros de "Juan Perez":
delete from libros

where codigoeditorial in
(select e.codigo
from editoriales as e
join libros
on codigoeditorial=e.codigo
where autor='Juan Perez');

```

La ejecución de este lote de comandos SQL genera una salida similar a:



The screenshot shows a PostgreSQL terminal window with the title bar 'bd1 on postgres@PostgreSQL 11'. The main area contains the following SQL code:

```

45
46 -- Eliminamos todos los libros de las editoriales que tiene publicados libros de "Juan Perez":
47 delete from libros
48 where codigoeditorial in
49 (select e.codigo
50   from editoriales as e
51   join libros
52     on codigoeditorial=e.codigo
53   where autor='Juan Perez');
54

```

Below the code, there is a navigation bar with tabs: Data Output, Explain, Messages, Notifications, and Query History. The 'Messages' tab is selected. The status bar at the bottom shows 'DELETE 4' and 'Query returned successfully in 195 msec.'

67 - Subconsulta (insert)

Aprendimos que una subconsulta puede estar dentro de un "select", "update" y "delete"; también puede estar dentro de un "insert".

Podemos ingresar registros en una tabla empleando un "select".

La sintaxis básica es la siguiente:

```
insert into TABLAENQUESEINGRESA (CAMPOSTABLA1)
select (CAMPOSTABLACONSULTADA)
from TABLACONSULTADA;
```

Un profesor almacena las notas de sus alumnos en una tabla llamada "alumnos". Tiene otra tabla llamada "aprobados", con algunos campos iguales a la tabla "alumnos" pero en ella solamente almacenará los alumnos que han aprobado el ciclo.

Ingresamos registros en la tabla "aprobados" seleccionando registros de la tabla "alumnos":

```
insert into aprobados (documento,nota)
select (documento,nota)
from alumnos;
```

Entonces, se puede insertar registros en una tabla con la salida devuelta por una consulta a otra tabla; para ello escribimos la consulta y le anteponemos "insert into" junto al nombre de la tabla en la cual ingresaremos los registros y los campos que se cargarán (si se ingresan todos los campos no es necesario listarlos).

La cantidad de columnas devueltas en la consulta debe ser la misma que la cantidad de campos a cargar en el "insert".

Se pueden insertar valores en una tabla con el resultado de una consulta que incluya cualquier tipo de "join".

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists alumnos;
drop table if exists aprobados;

create table alumnos(
    documento char(8) not null,
    nombre varchar(30),
```

```

nota decimal(4,2),
primary key(documento)

);

create table aprobados(
documento char(8) not null,
nota decimal(4,2),
primary key(documento)

);

insert into alumnos values('30000000','Ana Acosta',8);
insert into alumnos values('30111111','Betina Bustos',9);
insert into alumnos values('30222222','Carlos Caseros',2.5);
insert into alumnos values('30333333','Daniel Duarte',7.7);
insert into alumnos values('30444444','Estela Esper',3.4);

-- Ingresamos registros en la tabla "aprobados" seleccionando
-- registros de la tabla "alumnos":
insert into aprobados
select documento,nota
from alumnos
where nota>=4;

-- Veamos si los registros se han cargado:
select * from aprobados;

```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL:11
23 -- Ingresamos registros en la tabla "aprobados" seleccionando
24 -- registros de la tabla "alumnos":
25 insert into aprobados
26   select documento,nota
27     from alumnos
28    where nota>=4;
29
30 -- Veamos si los registros se han cargado:
31 select * from aprobados;
32
```

Data Output		Explain	Messages	Notifications	Query History
	documento character(8)	nota numeric(4,2)			
1	30000000	8.00			
2	30111111	9.00			
3	30333333	7.70			

68 - Vistas

Una vista es una alternativa para mostrar datos de varias tablas. Una vista es como una tabla virtual que almacena una consulta. Los datos accesibles a través de la vista no están almacenados en la base de datos como un objeto.

Entonces, una vista almacena una consulta como un objeto para utilizarse posteriormente. Las tablas consultadas en una vista se llaman tablas base. En general, se puede dar un nombre a cualquier consulta y almacenarla como una vista.

Una vista suele llamarse también tabla virtual porque los resultados que retorna y la manera de referenciarlas es la misma que para una tabla.

Las vistas permiten:

- ocultar información: permitiendo el acceso a algunos datos y manteniendo oculto el resto de la información que no se incluye en la vista. El usuario solo puede consultar la vista.
- simplificar la administración de los permisos de usuario: se pueden dar al usuario permisos para que solamente pueda acceder a los datos a través de vistas, en lugar de concederle permisos para acceder a ciertos campos, así se protegen las tablas base de cambios en su estructura.
- mejorar el rendimiento: se puede evitar tipar instrucciones repetidamente almacenando en una vista el resultado de una consulta compleja que incluya información de varias tablas.

Podemos crear vistas con: un subconjunto de registros y campos de una tabla; una unión de varias tablas; una combinación de varias tablas; un resumen estadístico de una tabla; un subconjunto de otra vista, combinación de vistas y tablas.

Una vista se define usando un "select".

La sintaxis básica parcial para crear una vista es la siguiente:

```
create view NOMBREVISTA as
```

```
    SENTENCIAS SELECT
```

```
    from TABLA;
```

El contenido de una vista se muestra con un "select":

```
select *from NOMBREVISTA;
```

En el siguiente ejemplo creamos la vista "vista_empleados", que es resultado de una combinación en la cual se muestran 4 campos:

```
create view vista_empleados as
```

```
    select (apellido||' '||e.nombre) as nombre, sexo,
```

```
s.nombre as seccion, cantidadhijos  
from empleados as e  
join secciones as s  
on codigo=seccion
```

Para ver la información contenida en la vista creada anteriormente tipeamos:

```
select *from vista_empleados;
```

Podemos realizar consultas a una vista como si se tratara de una tabla:

```
select seccion,count(*) as cantidad  
from vista_empleados;
```

Los nombres para vistas deben seguir las mismas reglas que cualquier identificador. Para distinguir una tabla de una vista podemos fijar una convención para darle nombres, por ejemplo, colocar el sufijo “vista” y luego el nombre de las tablas consultadas en ellas.

Los campos y expresiones de la consulta que define una vista DEBEN tener un nombre. Se debe colocar nombre de campo cuando es un campo calculado o si hay 2 campos con el mismo nombre. Note que en el ejemplo, al concatenar los campos "apellido" y "nombre" colocamos un alias; si no lo hubiésemos hecho aparecería un mensaje de error porque dicha expresión DEBE tener un encabezado, PostgreSQL no lo coloca por defecto.

Los nombres de los campos y expresiones de la consulta que define una vista DEBEN ser únicos (no puede haber dos campos o encabezados con igual nombre). Note que en la vista definida en el ejemplo, al campo "s.nombre" le colocamos un alias porque ya había un encabezado (el alias de la concatenación) llamado "nombre" y no pueden repetirse, si sucediera, aparecería un mensaje de error.

Otra sintaxis es la siguiente:

```
create view NOMBREVISTA (NOMBRESDEENCABEZADOS)  
as  
SENTENCIASSELECT  
from TABLA;
```

Creamos otra vista de "empleados" denominada "vista_empleados_ingreso" que almacena la cantidad de empleados por año:

```
create view vista_empleados_ingreso (fecha,cantidad)  
as  
select extract(year from fechaingreso),count(*)  
from empleados
```

```
group by extract(year from fechaingreso);
```

La diferencia es que se colocan entre paréntesis los encabezados de las columnas que aparecerán en la vista. Si no los colocamos y empleamos la sintaxis vista anteriormente, se emplean los nombres de los campos o alias (que en este caso habría que agregar) colocados en el "select" que define la vista. Los nombres que se colocan entre paréntesis deben ser tantos como los campos o expresiones que se definen en la vista.

Las vistas se crean en la base de datos activa.

Al crear una vista, PostgreSQL verifica que existan las tablas a las que se hacen referencia en ella.

Se aconseja probar la sentencia "select" con la cual definiremos la vista antes de crearla para asegurarnos que el resultado que retorna es el imaginado.

Se pueden construir vistas sobre otras vistas.

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
-- Borramos las tablas secciones y empleados si existen  
-- y si hay objetos dependientes como las vistas también las elimina  
drop table if exists secciones cascade;  
drop table if exists empleados cascade;
```

```
create table secciones(
```

```
codigo serial,
```

```
nombre varchar(20),
```

```
sueldo decimal(5,2),
```

```
primary key (codigo)
```

```
);
```

```
create table empleados(
```

```
legajo serial,
```

```
documento char(8),
```

```
sexo char(1),
```

```
apellido varchar(20),
```

```
nombre varchar(20),
```

```
domicilio varchar(30),
```

```

seccion smallint not null,
cantidadhijos smallint,
estadocivil char(10),
fechaingreso date,
primary key (legajo)

);

insert into secciones(nombre,sueldo) values('Administracion',300);
insert into secciones(nombre,sueldo) values('Contaduría',400);
insert into secciones(nombre,sueldo) values('Sistemas',500);

insert into empleados
(documento,sexo,apellido,nombre,domicilio,seccion,cantidadhijos,estadocivil,fechaingreso)
values('22222222','f','Lopez','Ana','Colon 123',1,2,'casado','1990-10-10');

insert into empleados
(documento,sexo,apellido,nombre,domicilio,seccion,cantidadhijos,estadocivil,fechaingreso)
values('23333333','m','Lopez','Luis','Sucre 235',1,0,'soltero','1990-02-10');

insert into empleados
(documento,sexo,apellido,nombre,domicilio,seccion,cantidadhijos,estadocivil,fechaingreso)
values('24444444','m','Garcia','Marcos','Sarmiento 1234',2,3,'divorciado','1998-07-12');

insert into empleados
(documento,sexo,apellido,nombre,domicilio,seccion,cantidadhijos,estadocivil,fechaingreso)
values('25555555','m','Gomez','Pablo','Bulnes 321',3,2,'casado','1998-10-09');

insert into empleados
(documento,sexo,apellido,nombre,domicilio,seccion,cantidadhijos,estadocivil,fechaingreso)
values('26666666','f','Perez','Laura','Peru 1254',3,3,'casado','2000-05-09');

-- Creamos la vista "vista_empleados", que es resultado de una combinación en la cual
-- se muestran 5 campos:

```

```
create view vista_empleados as  
select (apellido||' '||e.nombre) as nombre,sexo,  
s.nombre as seccion,cantidadhijos  
from empleados as e  
join secciones as s  
on codigo=seccion;
```

-- Vemos la información de la vista:

```
select * from vista_empleados;
```

-- Realizamos una consulta a la vista como si se tratara de una tabla:

```
select seccion,count(*) as cantidad  
from vista_empleados  
group by seccion;
```

-- Creamos otra vista de "empleados" denominada "vista_empleados_ingreso"

-- que almacena la cantidad de empleados por año:

```
create view vista_empleados_ingreso (fecha,cantidad)  
as  
select extract(year from fechaingreso),count(*)  
from empleados  
group by extract(year from fechaingreso);
```

-- Vemos la información:

```
select * from vista_empleados_ingreso;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL:11
63
64 -- Creamos otra vista de "empleados" denominada "vista_empleados_ingreso"
65 -- que almacena la cantidad de empleados por año:
66 create view vista_empleados_ingreso (fecha,cantidad)
67 as
68 select extract(year from fechaingreso),count(*)
69   from empleados
70   group by extract(year from fechaingreso);
71
72 -- Vemos la información:
73 select * from vista_empleados_ingreso;
74
```

Data Output Explain Messages Notifications Query History

	fecha	cantidad
1	2000	1
2	1998	2
3	1990	2

69 - Vistas (eliminar)

Para quitar una vista se emplea "drop view":

```
drop view NOMBREVISTA;
```

Si se intenta eliminar una tabla a la que hace referencia una vista, la tabla no se elimina, hay que eliminar la vista previamente.

Solo el propietario puede eliminar una vista.

Eliminamos la vista denominada "vista_empleados":

```
drop view vista_empleados;
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists empleados cascade;
```

```
drop table if exists secciones cascade;
```

```
create table secciones(
```

```
    codigo serial,
```

```
    nombre varchar(20),
```

```
    sueldo decimal(5,2),
```

```
    primary key (codigo)
```

```
);
```

```
create table empleados(
```

```
    legajo serial,
```

```
    documento char(8),
```

```
    sexo char(1),
```

```
    apellido varchar(20),
```

```
    nombre varchar(20),
```

```
    domicilio varchar(30),
```

```
    seccion smallint not null,
```

```

cantidadhijos smallint,
estadocivil char(10),
fechaingreso date,
primary key (legajo)
);

insert into secciones(nombre,sueldo) values('Administracion',300);
insert into secciones(nombre,sueldo) values('Contaduría',400);
insert into secciones(nombre,sueldo) values('Sistemas',500);

insert into empleados
(documento,sexo,apellido,nombre,domicilio,seccion,cantidadhijos,estadocivil,fechaingreso)
values('22222222','f','Lopez','Ana','Colon 123',1,2,'casado','1990-10-10');

insert into empleados
(documento,sexo,apellido,nombre,domicilio,seccion,cantidadhijos,estadocivil,fechaingreso)
values('23333333','m','Lopez','Luis','Sucre 235',1,0,'soltero','1990-02-10');

insert into empleados
(documento,sexo,apellido,nombre,domicilio,seccion,cantidadhijos,estadocivil,fechaingreso)
values('24444444','m','Garcia','Marcos','Sarmiento 1234',2,3,'divorciado','1998-07-12');

insert into empleados
(documento,sexo,apellido,nombre,domicilio,seccion,cantidadhijos,estadocivil,fechaingreso)
values('25555555','m','Gomez','Pablo','Bulnes 321',3,2,'casado','1998-10-09');

insert into empleados
(documento,sexo,apellido,nombre,domicilio,seccion,cantidadhijos,estadocivil,fechaingreso)
values('26666666','f','Perez','Laura','Peru 1254',3,3,'casado','2000-05-09');

```

-- Creamos la vista "vista_empleados", que es resultado de una combinación en la cual se muestran 5 campos:

```
create view vista_empleados as
```

```
select (apellido||' '|e.nombre) as nombre,sexo,  
s.nombre as seccion, cantidadhijos  
from empleados as e  
join secciones as s  
on codigo=seccion;
```

-- Veamos la información de la vista:

```
select * from vista_empleados;
```

-- Eliminamos la vista:

```
drop view vista_empleados;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL terminal window titled "bd1 on postgres@PostgreSQL 11". The command history (lines 52 to 59) contains the following SQL code:

```
52  
53 -- Veamos la información de la vista;  
54 select * from vista_empleados;  
55  
56 -- Eliminamos la vista;  
57 drop view vista_empleados;  
58  
59
```

Below the command history, there is a navigation bar with tabs: Data Output, Explain, Messages, Notifications, and Query History. The "Messages" tab is currently selected. The message area displays the output of the commands:

```
DROP VIEW  
  
Query returned successfully in 200 msec.
```

70 - Secuencias (create sequence- alter sequence - nextval - drop sequence)

Una secuencia (sequence) se emplea para generar valores enteros secuenciales únicos y asignárselos a campos numéricos; se utilizan generalmente para las claves primarias de las tablas garantizando que sus valores no se repitan (normalmente utilizamos la definición de un campo serial, este tiene asociado una secuencia en forma automática)

Una secuencia es una tabla con un campo numérico en el cual se almacena un valor y cada vez que se consulta, se incrementa tal valor para la próxima consulta.

Sintaxis general:

```
create sequence NOMBRESEQUENCIA
```

```
    start with VALORENTERO
```

```
    increment by VALORENTERO
```

```
    maxvalue VALORENTERO
```

```
    minvalue VALORENTERO
```

```
    cycle;
```

- La cláusula "start with" indica el valor desde el cual comenzará la generación de números secuenciales. Si no se especifica, se inicia con el valor que indique "minvalue".
- La cláusula "increment by" especifica el incremento, es decir, la diferencia entre los números de la secuencia; debe ser un valor numérico entero positivo o negativo diferente de 0. Si no se indica, por defecto es 1.
- "maxvalue" define el valor máximo para la secuencia. Si se omite, por defecto es 9223372036854775807.
- "minvalue" establece el valor mínimo de la secuencia. Si se omite será -9223372036854775808.
- La cláusula "cycle" indica que, cuando la secuencia llegue a máximo valor (valor de "maxvalue") se reinicie, comenzando con el mínimo valor ("minvalue") nuevamente, es decir, la secuencia vuelve a utilizar los números. Si se omite, por defecto la secuencia se crea "nocycle", lo que produce un error si supera el máximo valor.

Si no se especifica ninguna cláusula, excepto el nombre de la secuencia, por defecto, comenzará en 1, se incrementará en 1, el mínimo valor será -9223372036854775808, el máximo será 9223372036854775807 y "nocycle".

En el siguiente ejemplo creamos una secuencia llamada "sec_codigolibros", estableciendo que comience en 1, sus valores estén entre 1 y 99999 y se incrementen en 1, por defecto, será "nocycle":

```
create sequence sec_codigolibros  
start with 1  
increment by 1  
maxvalue 99999  
minvalue 1;
```

Si bien, las secuencias son independientes de las tablas, se utilizarán generalmente para una tabla específica, por lo tanto, es conveniente darle un nombre que refiera a la misma.

Otro ejemplo:

```
create sequence sec_numerosocios  
start with 2  
increment by 5  
cycle;
```

La secuencia anterior, "sec_numerosocios", incrementa sus valores en 5 y al llegar al máximo valor recomenzará la secuencia desde el valor mínimo.

Dijimos que las secuencias son tablas; por lo tanto se accede a ellas mediante consultas, empleando "select".

```
select * from sec_numerosocios;
```

Tenemos una función que nos retorna el próximo valor de la secuencia:

```
select nextval('sec_numerosocios');  
select nextval('sec_numerosocios');
```

Imprime un 2 y un 3.

El valor returned "nextval" pueden usarse cuando definimos una tabla.

Veamos un ejemplo completo:

Creamos una secuencia para el código de la tabla "libros", especificando el valor máximo, el incremento y que no sea circular:

```
create sequence sec_codigolibros  
minvalue 1000  
maxvalue 999999
```

increment by 1;

Creamos la tabla libros y asociamos a la columna codigo la secuencia sec_codigolibros:

```
create table libros(  
    codigo nextval('sec_codigolibros'),  
    titulo varchar(30),  
    autor varchar(30),  
    editorial varchar(15),  
    primary key (codigo)  
);
```

Ingresamos un registro en "libros":

```
insert into libros(titulo,autor,editorial) values  
('El aleph', 'Borges','Emece');
```

Ingresamos otro registro en "libros":

```
insert into libros(titulo,autor,editorial) values  
('Matematica estas ahi', 'Paenza','Nuevo siglo');
```

Luego si imprimimos los dos registros podemos comprobar que el campo codigo almacena el valor 1000 y 1001 respectivamente.

Para eliminar una secuencia empleamos "drop sequence". Sintaxis:

```
drop sequence NOMBRESECUENCIA;
```

Si la secuencia depende de otro objeto (en este caso una tabla) no se procede al borrado), debemos primero borrar la tabla y luego la secuencia, o utilizar (borra los objetos asociados a la secuencia):

```
drop sequence NOMBRESECUENCIA cascade;
```

Si la secuencia no existe aparecerá un mensaje indicando tal situación.

Podemos modificar una secuencia con la siguiente sintaxis:

```
alter sequence NOMBRESECUENCIA  
start with VALORENTERO  
increment by VALORENTERO  
maxvalue VALORENTERO  
minvalue VALORENTERO
```

```
cycle;
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros;
```

```
drop sequence if exists sec_codigolibros;
```

```
create sequence sec_codigolibros
```

```
minvalue 1000
```

```
maxvalue 999999
```

```
increment by 1;
```

```
-- Creamos la tabla libros y asociamos el valor por defecto
```

```
-- para el campo codigo el valor de la secuencia:
```

```
create table libros(
```

```
codigo bigint default nextval('sec_codigolibros'),
```

```
titulo varchar(30),
```

```
autor varchar(30),
```

```
editorial varchar(15),
```

```
primary key (codigo)
```

```
);
```

```
insert into libros(titulo,autor,editorial) values
```

```
('El aleph', 'Borges','Emece');
```

```
insert into libros(titulo,autor,editorial) values
```

```
('Matematica estas ahi', 'Paenza','Nuevo siglo');
```

```
select * from libros;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```

bd1 on postgres@PostgreSQL 11
1 drop table if exists libros;
2 drop sequence if exists sec_codigolibros;
3
4 create sequence sec_codigolibros
5   minvalue 1000
6   maxvalue 999999
7   increment by 1;
8
9 -- Creamos la tabla libros y asociamos el valor por defecto
10 -- para el campo codigo el valor de la secuencia:
11 create table libros(
12   codigo bigint default nextval('sec_codigolibros'),
13   titulo varchar(30),
14   autor varchar(30),
15   editorial varchar(15),
16   primary key (codigo)
17 );
18
19 insert into libros(titulo,autor,editorial) values
20 ('El aleph', 'Borges','Emece');
21 insert into libros(titulo,autor,editorial) values
22 ('Matematica estas ahí', 'Paenza','Nuevo siglo');
23
24 select * from libros;
25

```

Data Output	Explain	Messages	Notifications	Query History
codigo bigint	titulo character varying (30)	autor character varying (30)	editorial character varying (15)	
1	1000	El aleph	Borges	Emece
2	1001	Matematica estas ahí	Paenza	Nuevo siglo

71 - Funciones

Hemos visto que PostgreSQL nos brinda un conjunto de funciones para el manejo de fechas, string, números etc. pero además nos permite crear funciones propias.

La creación de una función es muy útil cuando queremos reutilizar un algoritmo. Podemos crear una función y luego llamarla en diferentes situaciones.

La sintaxis básica para crear una función es:

```
create or replace function [nombre de la función]([parámetros]) returns [tipo de dato que retorna]
```

as

[definición de la función]

language [lenguaje utilizado]

Se utiliza la sintaxis 'create or replace function' por si ya se creó la función con anterioridad (si no disponemos 'or replace' y la función ya existe aparecerá un mensaje de error)

La [definición de la función] depende del lenguaje utilizado para codificarla, pudiendo ser:

SQL

PL/PGSQL

PL/TCL

PL/Perl

C

Alguno de los valores anteriores debemos indicarlo en la directiva 'language'.

72 - Funciones SQL

La forma más fácil de implementar funciones es utilizar el lenguaje SQL. Una función SQL nos permite dar un nombre a uno o varios comandos sql.

Luego la sintaxis para implementar una función SQL:

```
create or replace function [nombre de la función]([parámetros]) returns [tipo de dato que retorna]
```

```
as
```

```
[comandos sql]
```

```
language sql
```

Como primer problema implementaremos una función que reciba dos enteros y retorne la suma de los mismos:

```
create or replace function sumar(integer,integer) returns integer
```

```
AS
```

```
'select $1+$2;'
```

```
language sql;
```

Cada parámetro se lo accede luego mediante la posición que ocupa y se le antecede el carácter \$. El o los comandos SQL deben ir entre simples comillas (si tenemos que utilizar las simples comillas en el comando SQL debemos disponer dos simples comillas seguidas) y separados por punto y coma. Luego indicamos al final que se trata de una función SQL.

Para llamar luego a esta función lo hacemos por ejemplo en un select:

```
select sumar(3,4);
```

Podemos acceder perfectamente a una o más tablas en la función. Confeccionaremos una función que acceda a la tabla usuarios:

```
create table usuarios (
    nombre varchar(30),
    clave varchar(10)
);
```

y rescate la clave de un usuario que le pasamos como parámetro:

```
create or replace function retornarclave(varchar) returns varchar
```

```
as
```

```
'select clave from usuarios where nombre=$1;'
```

```
language sql;
```

Luego para probar la función retornarclave debemos llamarla por ejemplo desde un select:

```
select retornarclave('Susana');
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
create or replace function sumar(integer,integer) returns integer
```

```
AS
```

```
'select $1+$2;'
```

```
language sql;
```

-- Llamamos la función que acabamos de crear:

```
select sumar(3,4);
```

```
drop table if exists usuarios;
```

-- Creamos la tabla usuarios:

```
create table usuarios (
```

```
    nombre varchar(30),
```

```
    clave varchar(10)
```

```
);
```

```
insert into usuarios (nombre, clave) values ('Marcelo','Boca');
```

```
insert into usuarios (nombre, clave) values ('JuanPerez','Juancito');
```

```
insert into usuarios (nombre, clave) values ('Susana','River');
```

```
insert into usuarios (nombre, clave) values ('Luis','River');
```

-- Creamos una función que reciba una cadena con el nombre de usuario

-- y retorne la clave de dicho usuario:

```
create or replace function retornarclave(varchar) returns varchar
```

as

```
'select clave from usuarios where nombre=$1;'
```

```
language sql;
```

-- Llamamos la función recuperando la clave del usuario llamada 'Susana':

```
select retornarclave('Susana');
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
1  create or replace function sumar(integer,integer) returns integer
2  AS
3      'select $1+$2;'
4  language sql;
5
6  -- Llamamos la función que acabamos de crear:
7  select sumar(3,4);
8
9  drop table if exists usuarios;
10
11 -- Creamos la tabla usuarios:
12 create table usuarios (
13     nombre varchar(30),
14     clave varchar(10)
15 );
16
17 insert into usuarios (nombre, clave) values ('Marcelo','Boca');
18 insert into usuarios (nombre, clave) values ('JuanPerez','Juancito');
19 insert into usuarios (nombre, clave) values ('Susana','River');
20 insert into usuarios (nombre, clave) values ('Luis','River');
21
22 -- Creamos una función que reciba una cadena con el nombre de usuario
23 -- y retorne la clave de dicho usuario:
24 create or replace function retornarclave(varchar) returns varchar
25 as
26     'select clave from usuarios where nombre=$1;'
27 language sql;
28
29 -- Llamamos la función recuperando la clave del usuario llamada 'Susana':
30 select retornarclave('Susana');
```

Data Output Explain Messages Notifications Query History

retornarclave
character varying

1 River

73 - Función SQL que no retorna dato (void)

Cuando queremos crear una función que no retorne dato lo debemos indicar luego de la palabra clave returns disponiendo el valor void:

```
create or replace function [nombre de la función]([parámetros]) returns void  
as  
[comandos sql]  
language sql
```

Confeccionaremos una función que cargue cuatro registros en la tabla usuarios:

```
create or replace function cargarusuarios() returns void  
as  
$$  
insert into usuarios (nombre, clave) values ('Marcelo','Boca');  
insert into usuarios (nombre, clave) values ('JuanPerez','Juancito');  
insert into usuarios (nombre, clave) values ('Susana','River');  
insert into usuarios (nombre, clave) values ('Luis','River');  
$$  
language sql;
```

Otra sintaxis permitida en PostgreSQL es encerrar los comando SQL entre dos símbolos de \$ (esto es muy útil si tenemos que utilizar la simple comilla dentro de los comandos SQL, en caso de disponer como delimitador las simples comillas la función debe expresarse:

```
create or replace function cargarusuarios() returns void  
as  
'  
insert into usuarios (nombre, clave) values ("Marcelo","Boca");  
insert into usuarios (nombre, clave) values ("JuanPerez","Juancito");  
insert into usuarios (nombre, clave) values ("Susana","River");  
insert into usuarios (nombre, clave) values ("Luis","River");  
'
```

```
language sql;
```

Debemos disponer dos comillas simples en lugar de una.

Luego para llamar la función lo hacemos:

```
select cargarusuarios();
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists usuarios;
```

```
create table usuarios (
```

```
    nombre varchar(30),
```

```
    clave varchar(10)
```

```
);
```

-- Definimos la función cargarusuarios:

```
create or replace function cargarusuarios() returns void
```

```
as
```

```
$$
```

```
insert into usuarios (nombre, clave) values ('Marcelo','Boca');
```

```
insert into usuarios (nombre, clave) values ('JuanPerez','Juancito');
```

```
insert into usuarios (nombre, clave) values ('Susana','River');
```

```
insert into usuarios (nombre, clave) values ('Luis','River');
```

```
$$
```

```
language sql;
```

-- Llamamos la función cargarusuarios:

```
select cargarusuarios();
```

-- Mostramos el contenido de la tabla usuarios:

```
select * from usuarios;
```

La ejecución de este lote de comandos SQL genera una salida similar a:

```
bd1 on postgres@PostgreSQL 11
1 drop table if exists usuarios;
2
3 create table usuarios (
4     nombre varchar(30),
5     clave varchar(10)
6 );
7
8 -- Definimos la función cargarusuarios:
9 create or replace function cargarusuarios() returns void
10 as
11 $$
12     insert into usuarios (nombre, clave) values ('Marcelo','Boca');
13     insert into usuarios (nombre, clave) values ('JuanPerez','Juancito');
14     insert into usuarios (nombre, clave) values ('Susana','River');
15     insert into usuarios (nombre, clave) values ('Luis','River');
16 $$
17 language sql;
18
19 -- Llamamos la función cargarusuarios:
20 select cargarusuarios();
21
22 -- Mostramos el contenido de la tabla usuarios:
23 select * from usuarios;
```

	nombre	clave
1	Marcelo	Boca
2	JuanPerez	Juancito
3	Susana	River
4	Luis	River

74 - Función SQL que retorna un dato compuesto.

Hemos visto que una función puede no retornar dato, retornar un dato simple (integer, varchar etc.) Ahora veremos como retornar toda una fila de una tabla.

Para indicar que una función retorna una fila de una tabla debemos indicar luego de la palabra returns el nombre de la tabla. Por ejemplo si creamos la tabla libros y cargamos algunos registros:

```
create table libros(  
    codigo serial,  
    titulo varchar(40) not null,  
    autor varchar(20) default 'Desconocido',  
    editorial varchar(20),  
    precio decimal(6,2),  
    primary key (codigo)  
);
```

```
insert into libros (titulo,autor,editorial,precio)  
values('El aleph','Borges','Emece',25.33);  
insert into libros (titulo,autor,editorial,precio)  
values('Java en 10 minutos','Mario Molina','Siglo XXI',50.65);  
insert into libros (titulo,autor,editorial,precio)  
values('Alicia en el pais de las maravillas','Lewis Carroll','Emece',19.95);  
insert into libros (titulo,autor,editorial,precio)  
values('Alicia en el pais de las maravillas','Lewis Carroll','Planeta',15);
```

Luego al definir la función indicamos el nombre de la tabla como dato que devuelve:

```
create or replace function retornarlibro(int) returns libros  
as  
'select * from libros where codigo=$1 ;'
```

```
language sql;
```

Debemos siempre disponer un comando SQL que retorne una fila de la tabla. Luego al llamar la función con un código de libro:

```
select retornarlibro(4);
```

Tenemos como resultado un dato compuesto con todos los campos de dicha fila:

```
(3,"Alicia en el pais de las maravillas","Lewis Carroll",Emece,19.95)
```

Ingresemos el siguiente lote de comandos SQL en Navicat:

```
drop table if exists libros cascade;
```

```
create table libros(
```

```
codigo serial,
```

```
titulo varchar(40) not null,
```

```
autor varchar(20) default 'Desconocido',
```

```
editorial varchar(20),
```

```
precio decimal(6,2),
```

```
primary key (codigo)
```

```
);
```

```
insert into libros (titulo,autor,editorial,precio)
```

```
values('El aleph','Borges','Emece',25.33);
```

```
insert into libros (titulo,autor,editorial,precio)
```

```
values('Java en 10 minutos','Mario Molina','Siglo XXI',50.65);
```

```
insert into libros (titulo,autor,editorial,precio)
```

```
values('Alicia en el pais de las maravillas','Lewis Carroll','Emece',19.95);
```

```
insert into libros (titulo,autor,editorial,precio)
```

```
values('Alicia en el pais de las maravillas','Lewis Carroll','Planeta',15);
```

```
-- Implementamos una función que reciba un código de libro y
```

```
-- retorne toda el registro que coincida con dicho código:
```

```
create or replace function retornarlibro(int) returns libros
```

```
as
```

```
'select * from libros where codigo=$1 ;'
```

```
language sql;
```

-- Llamamos a la función:

```
select retornarlibro(4);
```

La ejecución de este lote de comandos SQL genera una salida similar a:

The screenshot shows a PostgreSQL terminal window. The command history (psql) includes:

```
bd1 on postgres@PostgreSQL:11
22   -- Implementamos una función que reciba un código de libro y
23   -- retorne toda el registro que coincida con dicho código:
24   create or replace function retornarlibro(int) returns libros
25   as
26     'select * from libros where codigo=$1 ;'
27     language sql;
28
29   -- Llamamos a la función:
30   select retornarlibro(4);
```

The output section shows the results of the query:

Data Output	
	Data
1	{4,"Alicia en el país de las maravillas","Lewis Carroll",Planeta,15.00}