

El lenguaje SQL

En 1969, el investigador de IBM Edgar F. Codd definió el modelo de base de datos relacional, que se convirtió en la base para desarrollar el lenguaje SQL. Unos años más tarde, IBM comenzó a trabajar en un nuevo lenguaje para sistemas gestores de bases de datos relacionales basado en los hallazgos de Codd. El idioma originalmente se llamó SEQUEL, o lenguaje de consulta de inglés estructurado. El proyecto pasó por algunas implementaciones y revisiones, y el nombre del lenguaje cambió varias veces antes de finalmente aterrizar en SQL, pero se sigue pronunciando como el original.

El lenguaje de consulta estructurado (SQL) es el lenguaje de base de datos más popular en el ámbito mundial y fue declarado el lenguaje estándar de bases de datos relacionales por la ANSI (American National Standards Institute). El SQL se puede usar para compartir y gestionar datos, en particular los datos que se encuentran en los sistemas de bases de datos relacionales. Usando el SQL se puede consultar, actualizar y reorganizar los datos, así como crear y modificar el esquema (estructura) de un sistema de base de datos y controlar el acceso a los datos.

El objetivo del SQL es compilar y gestionar datos en grandes volúmenes, mucho mayores que los que se pueden manejar en una hoja electrónica. Si bien las hojas de cálculo pueden volverse complicadas con demasiada información que llena demasiadas celdas, las bases de datos SQL pueden manejar millones, o incluso miles de millones, de celdas de datos.

La mayoría de las organizaciones necesitan a alguien con conocimientos de SQL. Algunas posiciones que requieren habilidades de SQL incluyen:

- **Desarrollador de trasfondo (back-end):** una persona en esta posición administra el funcionamiento interno de las aplicaciones web, a diferencia de un desarrollador “front-end”, quien administra cómo se ve la aplicación y cómo funciona para los usuarios. Los desarrolladores de back-end trabajan debajo de las tablas del piso, por así decirlo, asegurándose de que la aplicación esté diseñada y funcionando correctamente.
- **El administrador de la base de datos (DBAs):** es una persona que se especializa en asegurarse de que los datos se almacenen y administren de manera adecuada y eficiente. Las bases de datos son más valiosas cuando permiten a los usuarios recuperar las combinaciones de datos deseadas de forma rápida y sencilla. Para ese trabajo, alguien debe asegurarse de que todos los datos se almacenen correctamente.
- **Analista de datos-** Alguien en esta posición analiza datos, quizás buscando tendencias relevantes en una organización en particular. A un analista se le puede presentar una pregunta determinada y encargarse de encontrar la

respuesta. Un ejemplo simple podría ser identificar cuáles clientes históricamente compran más durante el último trimestre de un año fiscal. Ese conocimiento permitiría a un departamento de ventas dirigirse eficientemente a los clientes en el momento adecuado.

- **Científico de datos:** esta es una posición muy similar a la de un analista de datos, pero los científicos de datos normalmente tienen la tarea de manejar datos en grandes volúmenes y acumularlos a velocidades mucho más altas.

El lenguaje SQL satisface necesidades académicas, profesionales y analíticas. Se usa tanto en computadores individuales, como en servidores corporativos. Con el progreso en la tecnología de base de datos, las aplicaciones basadas en SQL se han vuelto cada vez más asequibles para el usuario corriente. Esto se debe a la introducción de varios gestores de bases de datos relacionales de código abierto como MySQL, PostgreSQL, SQLite, Firebird y muchos más.

El estándar SQL ha pasado por muchos cambios durante los años, que han agregado una gran cantidad de nuevas funcionalidades al estándar, como soporte para el lenguaje XML, disparadores, coincidencia de expresiones regulares, consultas recursivas, secuencias estandarizadas y mucho más. Sin embargo, debe advertirse que existen pequeñas variaciones en el lenguaje SQL, dependiendo del sistema gestor de bases de datos utilizado. En este libro se considerará la sintaxis de PostgreSQL principalmente, aunque también se presentarán ejemplos en Hive del entorno para grandes volúmenes de datos de Hadoop. Por esto, en el capítulo siguiente, se presenta la forma de descargar e interactuar con bases de datos con estos dos gestores de bases de datos.

Elementos del lenguaje SQL

El lenguaje SQL se basa en varios elementos. Para comodidad de los desarrolladores de SQL, todos los comandos de lenguaje necesarios en los gestores de bases de datos correspondientes se ejecutan, generalmente, a través de una interfaz de línea de comandos de SQL específica. Los elementos que conforman el lenguaje SQL son:

- **Cláusulas:** las cláusulas son componentes o partes bien definidas de las declaraciones y las consultas.
- **Expresiones:** las expresiones pueden producir valores escalares o tablas, que consisten en columnas y filas de datos.
- **Predicados:** especifican condiciones, que se utilizan para limitar los efectos de las declaraciones o las consultas, o para cambiar el flujo de un programa.
- **Consultas:** una consulta recupera datos, según un criterio dado.
- **Declaraciones o sentencias:** con las declaraciones se pueden controlar las transacciones, el flujo de programas, las conexiones, las sesiones o los diagnósticos. En los sistemas de bases de datos, las sentencias SQL se utilizan para enviar consultas desde un programa cliente a un servidor donde reside la base de datos. En respuesta, el servidor procesa las sentencias de SQL y

devuelve las respuestas al programa cliente. Esto permite a los usuarios ejecutar una amplia gama de operaciones de manipulación de datos increíblemente rápidas, desde entradas de datos simples hasta consultas complicadas.

Es importante decir que las sentencias en el lenguaje SQL no son sensibles al caso. Se pueden escribir en minúsculas o en mayúsculas y el formato es libre. Esto significa que podemos escribir la declaración en una sola línea o en varias. Por esto, generalmente, una declaración finaliza con un punto y coma. También es importante añadir comentarios a las declaraciones o sentencias como parte de la documentación. Esto le permitirá al creador o a otro usuario con el mantenimiento de datos. Estos comentarios se pueden diferenciar por dos guiones seguidos, si solo ocupan una línea. Cuando son de varias líneas, se usan los símbolos `/*` al inicio y `*/` al final.

Consultas SQL

Las consultas son las operaciones más comunes y esenciales del lenguaje SQL, de ahí su nombre. A través de una consulta SQL, se puede buscar en la base de datos la información requerida. Las consultas SQL se ejecutan con la sentencia "SELECT". Una consulta SQL puede ser más específica, con la ayuda de varias cláusulas:

- FROM (DE)- indica la tabla donde se realizará la búsqueda.
- WHERE (DONDE): se utiliza para definir las filas, en las que se realizará la búsqueda. Se excluirán todas las filas para las cuales la cláusula WHERE no es verdadera.
- ORDER BY (ORDENAR POR): esta es la única forma de ordenar los resultados en SQL. De lo contrario, serán devueltos en el orden como se guardaron los datos.

Ejemplos de consultas en SQL:

```
SELECT Nombre, Apellidos  
FROM Empleados  
ORDER BY Apellidos, Nombre;
```

```
SELECT name as pais, population as poblacion  
FROM world  
ORDER BY name;
```

Como recién se dijo, las consultas son la parte esencial del lenguaje SQL el motivo de tener una base de datos y por eso, se debe entrar en muchos más detalles pero que ahora no es conveniente abordar pues primero se deben crear relaciones y poblarlas para hacer ejercicios a medida que se aprende.

El lenguaje SQL se subdivide en 3 sub-lenguajes: el lenguaje de definición de datos (data definition language o abreviadamente DDL) para crear, modificar o eliminar objetos de datos como las tablas o los índices, el lenguaje de manipulación de datos (data management language o abreviadamente DML) para insertar, actualizar o eliminar datos individuales y, por último el lenguaje de control (DCL) para el manejo concurrente y las transacciones en la base de datos. Por esto, a continuación presentaremos ahora las declaraciones del lenguaje SQL, de acuerdo con estos sub-lenguajes, profundizando más adelante en las consultas.

Lenguaje para la Definición de Datos (DDL)

El objeto principal del modelo relacional es guardar y acceder a los datos guardados en la base de datos, estructurados, en las relaciones o tablas. Para la creación de una tabla se emplea el comando CREATE TABLE, cuya sintaxis es:

```
CREATE TABLE <[esquema.]nombre_tabla>
    ({<atributo1> tipo [DEFAULT expr]
      [restricción de columna[, ...]]
      |restricción_tabla } [, ... ] );
```

Donde *tipo* define el tipo de dato para el atributo y puede ser, entre otros:

Char(n)	Cadena de n caracteres de longitud fija
Varchar(n)	Cadena de n caracteres de longitud variable
Number[(n,m)]	Números de n dígitos con m decimales
Date	Fechas
Long o text	Cadenas largas de caracteres, sin límite
RAW y LONG RAW	Datos binarios (programas, imágenes o sonidos...)

Al definir un tipo de datos para cada columna, se está declarando una restricción primaria sobre ella, buscando con ello la integridad de dominio. Esto es, los posibles valores que puede tomar. Aunque podemos limitar, aún más, el conjunto de posibles valores, adicionando una restricción o "constraint" dentro de la definición de la tabla.

Las restricciones más comunes para los atributos son:

- **NOT NULL.** Esta restricción asegura que una columna o atributo no puede ser nulo.
- **DEFAULT.** Ofrece un valor por defecto, cuando no se proporciona ninguno.
- **UNIQUE.** Asegura que todos los valores para una columna sean diferentes..
- **PRIMARY KEY.** Permite identificar inequívocamente a una fila o tupla en una relación o tabla básica.
- **CHECK.** Asegura que todos los valores para un atributo cumplen ciertas condiciones.

- **REFERENCES.** Asegura que el valor de un atributo existe en otra tabla que sirve de referencia. Es decir, permite velar por la integridad referencial.

Las anteriores restricciones pueden llevar un nombre con el objeto de saber cuál es el impedimento al ingresar o actualizar un dato. También, si se quiere modificar o eliminar dicha restricción. Ahora mostraremos unos ejemplos del uso del comando CREATE.

--Creación de la tabla CARRERAS cuya clave primaria es el atributo Id

```
CREATE TABLE carrera (id CHAR(2) PRIMARY KEY,  
                        nombre VARCHAR(30) NOT NULL UNIQUE);
```

/* Creación de la tabla de estudiantes. El atributo sexo sólo puede ser F o M, en minúsculas o mayúsculas, según la restricción de columna impuesta. También se especifica que *id_carrera* es clave foránea y por eso, no necesita la definición del tipo de dato */

```
CREATE TABLE estudiante (cc char(8) PRIMARY KEY,  
                          nombre VARCHAR2(15) NOT NULL,  
                          id_carrera NOT NULL REFERENCES carrera(id),  
                          fecha_ing DATE NOT NULL,  
                          tel VARCHAR(7),  
                          sexo CHAR(1) NOT NULL CONSTRAINT vlrs_sexo  
                          CHECK(UPPER(SEXO IN ('F','M'))));
```

/* Creación de la tabla de cursos. Como la clave primaria es compuesta, se debe poner una restricción de tabla, al final*/

```
CREATE TABLE curso (id CHAR(3),  
                    grupo CHAR(2),  
                    cc_prof VARCHAR(8),  
                    PRIMARY KEY (id, grupo));
```

/* La creación de la tabla de registros académicos, representa una entidad débil pues depende la existencia de un estudiante y de un curso. Como un estudiante puede repetir, la fecha también es parte de la clave primaria. Por eso, al final aparece la restricción de tabla PRIMARY KEY. Adicionalmente, se especifican tres claves foráneas, una como restricción de columna y otra como de tabla */

```
CREATE TABLE registro(cc NOT NULL REFERENCES estudiante(id),  
                      id_curso CHAR(3) NOT NULL,  
                      fecha DATE NOT NULL,  
                      grupo CHAR(2) NOT NULL,  
                      Nota NUMBER(3,1),  
                      PRIMARY KEY(cc, id_curso, grupo, fecha),  
                      FOREIGN KEY (id_curso, grupo) REFERENCES  
                      curso);
```

En la declaración anterior, se presentó un caso de una clave compuesta por cuatro atributos. Por eso, es muy recomendable, por simplicidad, crearle una clave artificial como un número consecutivo. Estos consecutivos o secuencias (sequences, en inglés) pueden ser generados y manejados por el propio sistema, en la mayoría de gestores de bases de datos. A continuación, se presenta un ejemplo.

```
CREATE TABLE producto (  
    nro SERIAL,  
    ...  
);
```

Por otro lado, también es posible crear una nueva tabla a partir de una ya existente. Esta característica es muy útil para hacer respaldos o para casos como el siguiente donde se quieren transferir datos de una tabla a otra, en forma masiva:

```
CREATE TABLE egresado AS  
    SELECT *  
    FROM estudiante  
    WHERE tdg ='aprobado';
```

Para el cambio en la estructura de una tabla se emplea el comando ALTER TABLE. Con este comando se pueden cambiar el tipo a una columna, borrarla o añadir otras. También sirve para agregar restricciones de columna o de tabla siempre que los datos almacenados lo permitan. La sintaxis es:

```
ALTER TABLE [esquema.]tabla  
    [ADD {      {columna tipo [DEFAULT expr] [restricción]...  
        |      ( { columna tipo [DEFAULT expr] [restricción]...  
            [, { columna tipo [DEFAULT expr] [restricción] ...  
                } ] ... ) } ]  
    [MODIFY { columna tipo [DEFAULT expr] [restricción]...  
        | columna tipo [DEFAULT expr] [restricción]...  
            [, (columna tipo [DEFAULT expr]  
                [restricción ...]) } ]  
    [DROP cláusula] ...
```

La instrucción ALTER TABLE utiliza, los elementos que se enuncian a continuación.

ADD

Agrega una nueva columna o restricción.

MODIFY

Modifica la definición de una columna ya existente.

DEFAULT

Especifica un valor por defecto para una nueva columna o nuevo valor por defecto para una columna ya existente.

Un valor por defecto no puede hacer referencia a otras columnas o a fechas completamente especificadas. También se puede usar la fecha del sistema que se obtiene con la función `now()`.

Restricción de columna

Añade o quita una restricción NOT NULL a una columna existente.

Restricción de tabla

Añade o quita una restricción a la tabla.

Con la instrucción `ALTER TABLE`, entonces, se puede modificar una tabla existente de distintas formas, como se describe a continuación.

A. Utilizando `ADD` para agregar una nueva columna (o varias al tiempo) o restricciones a una tabla.

Ejemplos

```
/* Se agrega la columna "obs" de tipo alfanumérico hasta con 25 caracteres a la tabla de empleados */
```

```
ALTER TABLE Empleados ADD obs VARCHAR(25);
```

```
/* Se agregan salario, con un valor por defecto de tres millones, y la columna "obs", a la tabla Empleados. Cuando son varias columnas se deben encerrar entre paréntesis. */
```

```
ALTER TABLE Empleados ADD (obs varchar2(25), salario NUMBER  
DEFAULT 3000000);
```

```
/* Se agrega una restricción a la fecha de ingreso para que se mayor a la especificada que corresponde a la fecha de creación del instituto educativo */
```

```
ALTER TABLE estudiante ADD CONSTRAINT ver_fecha  
check(fecha_ing >= '01-jun-2015');
```

```
/* Se agrega una restricción de clave foránea a la tabla Pedidos. La clave foránea se refiere a ID de la tabla Empleados */
```

```
ALTER TABLE Pedidos ADD CONSTRAINT RelPedidos FOREIGN KEY  
(idemp) REFERENCES Empleados(ID);
```

B. Utilizando `MODIFY` para cambiar el tipo de una columna (o varias al tiempo) a una tabla o modificar restricciones, entre otras modificaciones.

-- Cambia el tipo de la columna *obs* a texto de longitud ilimitada.

```
ALTER TABLE Empleados MODIFY obs text;
```

/* Impone la restricción de requerido para la columna "obs" (esto sólo se puede hacer si la tabla está vacía o las tuplas existentes todas tienen valor para la columna).

```
ALTER TABLE estudiante MODIFY obs NOT NULL;
```

/* Con ALTER se le puede cambiar el nombre a una tabla

```
ALTER TABLE clientes RENAME TO cliente;
```

/* También se le puede cambiar el nombre a una columna con el comando ALTER

```
ALTER TABLE clientes RENAME direccioncli TO dir;
```

Utilizando la cláusula DROP solo se puede eliminar una restricción existente. No se pueden eliminar columnas. Para ello, se requiere el uso de tabla temporal donde se lleven los valores de los atributos de interés, luego se borre la tabla original y luego renombrar la temporal.

```
ALTER TABLE Pedidos DROP CONSTRAINT Relpedidos;
```

```
ALTER TABLE estudiante DROP CONSTRAINT ver_fecha;
```

Con la instrucción DROP se elimina una tabla existente de una base de datos.

Sintaxis

```
DROP [esquema.]tabla  
    [CASCADE CONSTRAINTS]
```

La opción CASCADE CONSTRAINTS elimina también todas las restricciones de integridad referencial que se refieran a la clave primaria de la tabla borrada. Si se omite la opción y existen tales restricciones, se devuelve un error y no se borra la tabla.

Ejemplo:

--Como la tabla está referenciada en la tabla estudiante, aparece un error

```
DROP TABLE carrera;
```

La orden, entonces, debe ser como aparece a continuación.

```
DROP TABLE carrera CASCADE CONSTRAINTS;
```


Ordenes de Manipulación de Datos (DML)

Las declaraciones para manipular datos permiten ingresar nuevas tuplas a las relaciones, actualizarlas o borrarlas. Estas declaraciones permiten que se afecten conjuntos de tuplas al tiempo. Por esto, los operadores DML son mucho más potentes que los operadores de los lenguajes tradicionales imperativos.

La inserción de datos puede hacerse en forma individual o realizar cargas masivas de datos al permitir una subconsulta dentro de la sentencia de inserción. También es posible ingresar datos a una tabla en forma indirecta a través de una vista, recordando que una vista es una tabla virtual originada a partir de una consulta. Se mostrarán ejemplos de estas posibilidades.

Para el ingreso de filas a una tabla o vista se emplea el comando INSERT, su sintaxis es:

```
INSERT INTO <nombre-tabla o vista> [atrib1, atrib2,... atribn]
      {VALUES (valor1, valor2,... valorn) | subconsulta}
```

En este comando, *valor_i* debe ir encerrado entre comillas sencillas si corresponde a un dato de tipo alfanumérico o fecha. Se puede omitir el nombre de las columnas si la lista de valores está en el mismo orden como se creó la tabla y tiene el mismo número de elementos que las columnas en la tabla. Debe decirse, además, que *valor_i* también puede ser una expresión que se pueda calcular en el tiempo de ejecución de una declaración de inserción.

Ejemplos:

-- En la declaración siguiente se especifican los nombres de las columnas

```
INSERT INTO registro (cod_asig, cc, nota) VALUES
      ('S003', '1893', Null);
```

/* En la declaración no se especifican los nombres de las columnas y los datos se deben entrar en el orden de creación de la tabla ESTUDIANTE */

```
INSERT INTO estudiante VALUES
      ('001', 'LUZ ADRIANA', 11, '09-FEB-2001',
      '2345678', 'F');
```

/* En este ejemplo se usa la cláusula DEFAULT para la fecha de creación, en lugar de entrar un valor: */

```
INSERT INTO pelicula VALUES
      ('UA502', 'Roma', DEFAULT, 'Drama', '82 minutos');
```

-- Para insertar varias tuplas al tiempo:

```
INSERT INTO pelicula (id, titulo, id_productor, fecha_prod,
tipo) VALUES
      ('B6717', 'El vecino marte', 110, now(), 'Ficción'),
```

```
('HG120', 'Grandes mentiritas', 140, DEFAULT, Comedia');
```

También se pueden ingresar varios registros o filas al tiempo, por medio de una subconsulta, así:

```
INSERT INTO egresados (SELECT * FROM estudiante WHERE  
                        fecha_grado IS NOT NULL);
```

Para reutilizar un comando de inserción, en Oracle, se pueden utilizar variables de ambiente de corto plazo, precedidas por un ampersand, o de largo plazo que se distinguen con doble ampersand. Las de corto plazo no conservan el valor para la próxima ejecución y por eso, lo pide nuevamente en cada ejecución. Las de largo plazo, en cambio, conservan el valor hasta salir de la sesión. Sólo piden el valor para la variable en la primera ejecución.

```
INSERT INTO estudiante VALUES  
( '&identificacion', '&nombre_completo', '&&id_carrera',  
  '&&fecha_ingreso', '&tel', upper('&sexo'), 'obs' );
```

Ya se presentó el comando INSERT del lenguaje DML para el ingreso de nuevas tuplas a una relación de la base de datos. Ahora, para modificar o cambiar datos en una tupla o un grupo de tuplas, se usa la orden UPDATE cuya sintaxis es:

```
UPDATE [esquema.]{tabla | vista}  
  SET {(columna [, columna] ... ) = (subconsulta)  
      | columna = {expresión | (subconsulta)}}  
  [,   {(columna [, columna] ... ) = (subconsulta)  
      | columna = {expresión | (subconsulta) } } ] ...  
[WHERE condición]
```

Ejemplos:

```
/* Para cambiar el teléfono de un estudiante, en particular */
```

```
UPDATE estudiante SET tel = '3052656363'  
WHERE cc = '8241003';
```

```
/* Se pueden actualizar varias columnas al tiempo. Se separan con comas.
```

```
UPDATE estudiante SET tel = '3052652020',  
                    barrio = 'Robledo',  
                    dir = 'Carrera 81 Nro 65-28'  
WHERE cc = '8241003';
```

Se debe tener precaución pues si un comando UPDATE no lleva la cláusula WHERE, afectará todas las filas de la tabla.

/* Aumenta las notas definitivas de todos los estudiantes, en todos los cursos */

```
UPDATE registro SET nota = nota + 0.5;
```

Para eliminar una fila o un conjunto de filas, se usa la orden DELETE cuya sintaxis es:

```
DELETE [FROM] <nombre-tabla>  
[WHERE condición]
```

Ejemplos

/* Elimina al empleado cuyo identificador es 7

```
DELETE FROM empleado WHERE id = 7;
```

/* Elimina a todos los empleados

```
DELETE FROM empleado;
```

El mecanismo que se utiliza para traer o consultar datos de la base de datos es el comando SELECT que se describió antes, sin mucho detalle. Es la orden más importante y compleja, que también puede ser parte de una orden INSERT, UPDATE o DELETE.

La orden SELECT tiene unas cláusulas que permiten seleccionar columnas y filas de una o más tablas o vistas, especificar una o más condiciones de filtrado, ordenar y obtener estadísticas de resumen. Su sintaxis más completa es

```
SELECT lista de expresiones  
[FROM lista de tablas]  
[WHERE condiciones]  
[ORDER BY atributo1 [DESC][, atributo2 [DESC],...]  
[GROUP BY atributo3, {atributo4...  
[HAVING condición de grupo]];
```

La lista de expresiones del SELECT puede ser cualquier combinación de lo siguiente:

- El asterisco (*) que indica todos los atributos de la tabla. Los atributos son mostrados en el orden que fueron listados cuando se crearon las tablas. Otra forma alternativa del asterisco es nombre-tabla.*.
- Nombre(s) de atributo(s): cualquier combinación de atributos encontrados en la lista de tablas de la cláusula FROM. Los atributos son mostrados en el orden en que son listados. Los nombres de los atributos son separados por comas y pueden recibir un alias.
- Expresiones aritméticas: se pueden utilizar los operadores aritméticos para calcular valores en tiempo de ejecución.

En la cláusula FROM de una consulta SELECT se especifica la lista de tablas y vistas que contienen los datos que se desean visualizar. En algunos sistemas gestores de bases de datos como PostgreSQL esta cláusula no es obligatoria pues sólo se quiere un cálculo o el valor de una función. Por ejemplo, la sentencia siguiente muestra la fecha actual como estampilla del tiempo (la fecha más la hora, los minutos y los segundos) y la raíz cuadrada de 2:

```
SELECT now() fecha, sqrt(4) raíz_dos;
```

En la anterior sentencia se usaron alias o nombres temporales (fecha y raíz_dos) para las columnas en la tabla resultante. En esa sentencia también se pudieron usar algunas variables del sistema para saber la fecha o la hora actual, como en esta otra orden:

```
SELECT CURRENT_FROM TIMESTAMP fecha_completa_con_hora,  
CURRENT_DATE fecha_actual, CURRENT_TIME hora_actual;
```

Por otro lado, en la cláusula opcional WHERE de una consulta, aparecerían todas las condiciones requeridas para dejar por fuera las tuplas indeseadas, que no cumplan con las condiciones especificadas. La cláusula ORDER BY indica que la tabla resultante será ordenada de acuerdo con el atributo o los atributos, allí especificados. En el siguiente ejemplo, se quiere una proyección de la descripción y del precio unitario de los productos, ordenados por la descripción, pero no se usó la cláusula WHERE. En el otro ejemplo si se puso la condición que el el precio unitario fuera mayor a los 10 dólares.

<pre>SELECT descripcion, preciounit FROM PRODUCTOS ORDER BY descripcion;</pre>		
<		
Output pane		
Data Output	Explain	Messages
History		
	descripcion character(50)	preciounit numeric
1	BASE DE MAQUILLAJE	14.70
2	BOTON PARA ASADO	4.70
3	CREMA DE LECHE	3.60
4	JAMON DE POLLO	2.80
5	LECHE CHOCOLATE	1.60
6	RIMMEL	12.90
7	SALAMI DE AJO	3.60
8	SALCHICHAS DE POLLO	2.90
9	SALCHICHAS VIENESAS	2.60
10	SOMBRA DE OJOS	9.80
11	YOGURT DE SABORES	1.60
12	YOGURT NATURAL	4.30

```
SELECT descripcion, preciounit FROM productos WHERE preciounit > 10  
ORDER BY descripcion;
```

	descripcion character(50)	preciounit numeric
1	BASE DE MAQUILLAJE	14.70
2	RIMMEL	12.90

Operadores lógicos relacionales

En SQL, se pueden usar los siguientes operadores lógicos y relacionales que se utilizan en la cláusula WHERE de una orden para determinar si una expresión es cierta o falsa y así excluir algunas filas o tuplas cuya evaluación dé falso.

Las condiciones de filtrado pueden ser varia y no una sola. Por esto, se pueden usar las dos conectivas lógicas: AND y OR.

=	Igual
<>	Diferente
<	Menor que
>	Mayor que
<=	Menor o igual
>=	Mayor o igual
LIKE	Como (parecido a)
IN	En el conjunto
NOT	Negación de una expresión
AND	Conjunción
OR	Disyunción
BETWEEN	Entre un rango de valores

El operador diferente en algunos sistemas gestores, también puede representarse con !=. Los operadores de mayor que, menor que también pueden usarse para cadenas de caracteres considerando el orden que prima convencionalmente: primero los números y luego las letras en orden lexicográfico.

Por su lado, el operador LIKE es muy útil para encontrar patrones. En el siguiente ejemplo, se muestra la orden para traer el precio unitario y la categoría para los productos cuya descripción contiene la palabra “pollo” en cualquier parte. Esto porque se usó el símbolo % que es un comodín para significar cualquier texto en la cadena y se puso antes y después del literal “POLLO”.

```
SELECT descripcion, preciounit, categoriaid FROM PRODUCTOS WHERE descripcion LIKE '%$POLLO$';
```

Output pane

	descripcion character(50)	preciounit numeric	categoriaid integer
1	SALCHICHAS DE POLLO	2.90	100
2	JAMON DE POLLO	2.80	100

En el próximo ejemplo, se muestra una sentencia donde se usan las conectivas AND y OR para especificar varias condiciones de filtrado. En la sentencia, se desea ver todos los atributos de los productos cuyo precio sea menor o igual a 10 dólares y que el código de su categoría sea 100 o 200.

```
SELECT * FROM PRODUCTOS WHERE preciounit <=10 AND (categoriaid = 100 OR categoriaid = 200);
```

Output pane

	productoid integer	proveedorid integer	categoriaid integer	descripcion character(50)	preciounit numeric	existencia integer
1	1	10	100	SALCHICHAS VIENESAS	2.60	200
2	2	10	100	SALAMI DE AJO	3.60	300
3	3	10	100	BOTON PARA ASADO	4.70	400
4	4	20	100	SALCHICHAS DE POLLO	2.90	200
5	5	20	100	JAMON DE POLLO	2.80	100
6	6	30	200	YOGURT NATURAL	4.30	80
7	7	30	200	LECHE CHOCOLATE	1.60	90
8	8	40	200	YOGURT DE SABORES	1.60	200
9	9	40	200	CREMA DE LECHE	3.60	30

En el próximo ejemplo, se hará uso del operador BETWEEN para ver los datos de los productos cuyo precio de venta esté entre 3 y 10 dólares, inclusive.

```
SELECT * FROM PRODUCTOS WHERE preciounit BETWEEN 3 AND 10;
```

Output pane

	productoid integer	proveedorid integer	categoriaid integer	descripcion character(50)	preciounit numeric	existencia integer
1	2	10	100	SALAMI DE AJO	3.60	300
2	3	10	100	BOTON PARA ASADO	4.70	400
3	6	30	200	YOGURT NATURAL	4.30	80
4	9	40	200	CREMA DE LECHE	3.60	30
5	13	60	600	SOMBRA DE OJOS	9.80	100

Cuando se quiera utilizar el valor nulo en una condición, se debe preguntar con el operador IS [NOT] NULL. No se debe usar el símbolo de igual o diferente pues nulo no es un valor. Acá, primero se muestra un ejemplo con el error, lo cual arroja una tabla vacía y luego se presenta la sentencia apropiada.

<code>SELECT nombre,nombrecontacto FROM clientes WHERE fax = NULL;</code>		
<		
Output pane		
Data Output Explain Messages History		
	nombre character(30)	nombrecontacto character(50)
<code>SELECT nombre,nombrecontacto FROM clientes WHERE fax IS NULL;</code>		
<		
Output pane		
Data Output Explain Messages History		
	nombre character(30)	nombrecontacto character(50)
1	SUPERMERCADO ESTRELLA	JUAN ALBAN
2	EL ROSADO	MARIA CORDERO
3	DISTRIBUIDORA PRENSA	PEDRO PINTO
4	SU TIENDA	PABLO PONCE
5	SUPERMERCADO DORADO	LORENA PAZ
6	MI COMISARIATO	ROSARIO UTRERAS
7	SUPERMERCADO DESCUENTO	LETICIA ORTEGA
8	EL DESCUENTO	JUAN TORRES
9	DE LUISE	JORGE PARRA
10	YARBANTRELLA	PABLO POLIT

Si se quisieran ver las categorías de los productos que se ofrecen actualmente, daríamos la orden siguiente:

```
SELECT categoriaid FROM productos
```

Output		Explain	Messages	History
categoriaid	integer			
100				
100				
100				
100				
100				
200				
200				
200				
200				
600				
600				
600				

Como se ve cada código de categoría se repite por cada producto que pertenece a dicha categoría. Para evitar esto, se utiliza la palabra **DISTINCT**:

```
SELECT DISTINCT categoriaid FROM productos
```

Output		Explain	Messages	History
categoriaid	integer			
200				
600				
100				

En ocasiones, se desea limitar los resultados a un número determinado de filas o tuplas. Para ellos se utiliza la palabra **LIMIT**, como en el siguiente ejemplo:

```
/* Muestre los tres productos más caros */
```

```
SELECT descripcion, preciounit
FROM productos
ORDER BY preciounit DESC
LIMIT 3;
```

Output		Explain	Messages	History
descripcion	character(50)	preciounit	numeric	
BASE DE MAQUILLAJE		14.70		
RIMMEL		12.90		
SOMBRA DE OJOS		9.80		

Ya vistos algunos ejemplos de los operadores y conectivas lógicas que podemos emplear en cualquier sentencia SQL, presentaremos las funciones predefinidas que también se pueden utilizar. Cabe señalar que el usuario, con el uso de un lenguaje de programación procedimental que también ofrecen la mayoría los sistemas gestores de base de datos puede crear funciones propias. Se pueden utilizar tanto las funciones predefinidas, como las demás definidas por los usuarios especialistas.

Funciones Matemáticas.

Estas funciones son aplicables a atributos o valores numéricos. Los sistemas gestores de bases de datos ofrecen una gran cantidad de funciones matemáticas predefinidas. En la tabla siguiente se muestran algunas funciones y luego se aplican, como ejemplo, para comprender mejor la funcionalidad.

ABS(X)	Devuelve el valor absoluto de un número
CEIL(X)	Si X es un decimal será redondeado al entero mayor (su techo).
FLOOR(X)	Si X es un decimal será redondeado al entero menor (su piso)
GREATEST(X,Y)	Devuelve el mayor entre x e y.
LEAST(X,Y)	Devuelve el menor valor entre x e y.
MOD(X,Y)	Devuelve el residuo de X / Y.
POWER(X,Y)	Devuelve X elevado a la Y.
ROUND(X,Y)	Redondea X a Y lugares decimales. Si se omite a Y, entonces redondea al entero más próximo.
SIGN(X)	Devuelve un signo menos si X < 0, en otro caso devuelve un más.
SQRT(X)	Devuelve la raíz cuadrada de x.

```
SELECT GREATEST(9,4) mayor, ABS(-1) absoluto,
       CEIL(3.4) cielo, FLOOR(2.8) piso, LEAST(9,4) menor,
       MOD(7,3) residuo, POWER(3,4) potencia, ROUND(2.99, 1)
       redondeo, SIGN(-4) signo, SQRT(25)raiz
```

La sentencia anterior, da como resultado lo siguiente:

Output pane										
Data Output Explain Messages History										
	mayor integer	absoluto integer	cielo numeric	piso numeric	menor integer	modulo integer	potencia double precision	redondeo numeric	signo double precision	raiz double precision
1	9	1	4	2	4	1	81	3.0	-1	5

Usando cualquier expresión en la proyección, que es la lista del SELECT, podemos ver los efectos de un aumento en los precios o el valor a cobrar por un producto después de un

descuento, por ejemplo. En la siguiente sentencia se muestra cada producto con su precio unitario y con el precio aumentado en un seis por ciento, y redondeado a dos decimales pues eso es suficiente cuando se habla de dólares.

```
SELECT descripcion, preciounit, round(preciounit*1.06, 2) precio_aumentado FROM productos
```

Output pane

	descripcion character(50)	preciounit numeric	precio_aumentado numeric
1	SALCHICHAS VIENESAS	2.60	2.76
2	SALAMI DE AJO	3.60	3.82
3	BOTON PARA ASADO	4.70	4.98
4	SALCHICHAS DE POLLO	2.90	3.07
5	JAMON DE POLLO	2.80	2.97
6	YOGURT NATURAL	4.30	4.56
7	LECHE CHOCOLATE	1.60	1.70
8	YOGURT DE SABORES	1.60	1.70
9	CREMA DE LECHE	3.60	3.82
10	BASE DE MAQUILLAJE	14.70	15.58
11	RIMMEL	12.90	13.67
12	SOMBRA DE OJOS	9.80	10.39

En este punto, es bueno señalar que todas las expresiones o funciones pueden usarse en sentencias diferentes a la consulta:

```
/* Se aumenta el salario de los empleados en un 10 por ciento y se redondea a cero decimales. */
```

```
UPDATE empleados SET salario = round(salario*1.1);
```

Funciones de cadenas de caracteres

Las funciones son bastante útiles, en especial cuando se desea realizar una consulta y no se conoce un dato en forma precisa, se quiere utilizar un patrón en ella o se quieren estandarizar los datos para hacer análisis o mejorar la presentación de los informes.

LEFT(<string>,X)	Devuelve los primeros X caracteres de la cadena.
RIGHT(<string>,X)	Devuelve los X caracteres que están al extremo derecho de la cadena.
UPPER(<string>)	Convierte la cadena de caracteres a mayúsculas
LOWER(<string>)	Convierte la cadena de caracteres a minúsculas
INITCAP(<string>)	Convierte la cadena de caracteres a Título (primer carácter en mayúsculas, el resto minúsculas)
LENGTH(<string>)	Devuelve el número de caracteres en la cadena.

<string> <string>	Concatena dos cadenas de caracteres.
LPAD(<string>,X,'*')	Rellena a la izquierda con * (o con cualquier carácter especificado entre comillas sencillas), para que la cadena quede con X caracteres.
RPAD(<string>,X,'*')	Rellena a la derecha con * (o con cualquier carácter especificado entre comillas sencillas), para que la cadena quede con X caracteres.
SUBSTR(<string>,X,Y)	Extrae Y caracteres de la cadena comenzando en la posición X.

En la siguiente declaración vamos a usar valores específicos, como se hizo con las funciones aritméticas, para mostrar el uso de las funciones de cadenas de caracteres.

```
SELECT LEFT('Margarita',4) cuatro_izq, RIGHT('Margarita',4)
       cuatro_der, UPPER('cambio'), LOWER('COLOMBIA'),
       INITCAP('inicial'), LENGTH('Margarita'),
       'Margarita' || ' ' || 'Gómez', LPAD('MA',8,'A'),
       RPAD('MA',8,'A'), SUBSTR('Margarita',4,10)
```

La sentencia anterior, da como resultado la tabla que aparece a continuación.

Output pane										
Data Output Explain Messages History										
	cuatro_izq text	cuatro_der text	upper text	lower text	initcap text	length integer	?column? text	lpad text	rpad text	substr text
1	Marg	rita	CAMBIO	colombia	Inicial	9	Margarita Gómez	AAAAAAMA	MAAAAAA	garita

La sentencia siguiente convierte dos atributos de tipo cadena de caracteres de la tabla de clientes, el nombre del contacto y el nombre de la empresa, en uno solo. Esto se logra con las dos barras que significan concatenación de cadenas o strings.

SELECT nombrecontacto ' ES EL CONTACTO DE LA EMPRESA ' nombre FROM clientes	
ane	
Output Explain Messages History	
?column? text	
JUAN ALBAN ES EL CONTACTO DE LA EMPRESA SUPERMERCADO ESTRELLA	
MARIA CORDERO ES EL CONTACTO DE LA EMPRESA EL ROSADO	
PEDRO PINTO ES EL CONTACTO DE LA EMPRESA DISTRIBUIDORA PRENSA	
PABLO PONCE ES EL CONTACTO DE LA EMPRESA SU TIENDA	
LORENA PAZ ES EL CONTACTO DE LA EMPRESA SUPERMERCADO DORADO	
ROSARIO UTRERAS ES EL CONTACTO DE LA EMPRESA MI COMISARIATO	
LETICIA ORTEGA ES EL CONTACTO DE LA EMPRESA SUPERMERCADO DESCUENTO	
JUAN TORRES ES EL CONTACTO DE LA EMPRESA EL DESCUENTO	
JORGE PARRA ES EL CONTACTO DE LA EMPRESA DE LUISE	
PABLO POLIT ES EL CONTACTO DE LA EMPRESA YARBANTRELLA	

Funciones para fechas

En el análisis de los datos es muy importante conocer estas funciones, pues permiten saber cosas como cuánto se demora la organización en despachar sus productos a los clientes de otras ciudades, por ejemplo. Esto, sin considerar su importancia en el análisis de series de tiempo. Este tipo de funciones también es muy importante para saber la edad de una persona, por ejemplo, a partir de la fecha de nacimiento

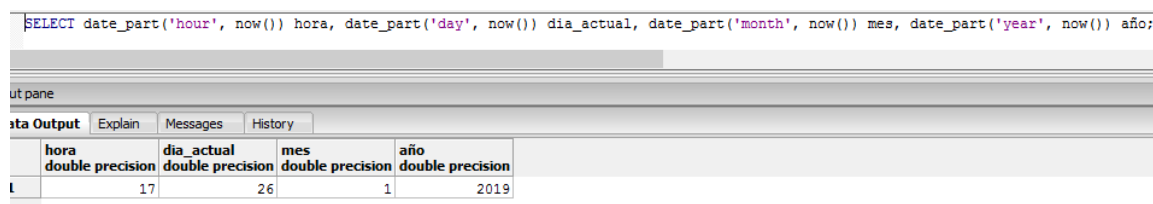
Generalmente, una fecha está definida mediante un tipo de dato estampilla en el tiempo (timestamp, en inglés). Esto significa que un dato de tipo fecha incluye no sólo la fecha, sino la hora con minutos y segundos. Por esto, en ocasiones es importante poder extraer parte de la fecha como el día o el mes. También, se puede querer truncar una fecha para no considerar el momento en el tiempo.

DATE_PART('parte', fecha)	Devuelve un número que corresponde a la parte de la fecha que se entra como argumento. La parte puede ser 'year', 'month', 'day', 'hour'
DATE_TRUNC('parte', fecha)	Se trunca la fecha desde la parte especificada (la hora, por ejemplo). Esto significa que se ponen ceros a partir de ahí.
AGE(estampilla de tiempo)	Devuelve el tiempo transcurrido desde la fecha entrada hasta la fecha actual.

En el ejemplo siguiente vamos a usar la función predefinida que nos dice la fecha actual en PostgreSQL que es *now()* para ver el uso de la función *date_part*:

```
SELECT date_part('hour', now()) hora, date_part('day', now()) dia_actual, date_part('month', now()) mes, date_part('year', now()) año;
```

Al ejecutar la anterior sentencia, vemos que los resultados todos son números.



The screenshot shows a PostgreSQL query execution window. The query is: `SELECT date_part('hour', now()) hora, date_part('day', now()) dia_actual, date_part('month', now()) mes, date_part('year', now()) año;`. The results are displayed in a table with the following data:

hora	dia_actual	mes	año
17	26	1	2019

El ejemplo siguiente muestra todos los atributos de las órdenes de compra hechas en el año 2007. Para ello, se usó la función *date_part*.

```
SELECT * FROM ordenes WHERE date_part('year', fechaorden) = 2007
```

Output pane

	ordenid integer	empleadoid integer	clienteid integer	fechaorden date	descuento integer
1	1	3	4	2007-06-17	5
2	2	3	4	2007-06-02	10
3	3	4	5	2007-06-05	6
4	4	2	6	2007-06-06	2
5	5	2	7	2007-06-09	
6	6	4	5	2007-06-12	10
7	7	2	5	2007-06-14	10
8	8	3	2	2007-06-13	10
9	9	3	2	2007-06-17	3
10	10	2	2	2007-06-18	2

En el siguiente ejemplo no es necesaria la función *date_part* pues se pregunta por una fecha específica:

```
SELECT * FROM ordenes WHERE fechaorden = '2007-06-17'
```

Output pane

	ordenid integer	empleadoid integer	clienteid integer	fechaorden date	descuento integer
1	1	3	4	2007-06-17	5
2	9	3	2	2007-06-17	3

Con la siguiente sentencia, se trunca la fecha dada sólo para ver hasta el día:

```
SELECT date_trunc('hour', TIMESTAMP '2019-01-16 20:38:40');
```

Si se quiere saber cuánto tiempo ha pasado desde una fecha determinada, se usa la función *age* como en el ejemplo que se muestra a continuación.

```
SELECT age(timestamp '1998-08-21')
```

Output pane

age interval
20 years 5 mons 5 days

La cláusula CASE

Esta cláusula es similar a la que ofrecen los lenguajes de programación para definir una **expresión condicional** o distintos flujos de acuerdo con el valor de la condición. La sintaxis es:

```
CASE WHEN condición THEN resultado
      [WHEN...]
      [ELSE resultado]
END
```

En la cláusula CASE, cada condición evalúa a cierto o a falso. Si la condición es cierta, entonces el valor de la expresión es el resultado que sigue a la condición. Si es falsa, entonces se sigue evaluando hasta que la condición sea cierta. Si ninguna se cumple, entonces el resultado es el que le sigue al ELSE. Por ejemplo:

```
SELECT * FROM prueba;
```

```
a
---
1
2
3
4
```

```
SELECT a,
       CASE WHEN a=1 THEN 'uno'
            WHEN a=2 THEN 'dos'
            ELSE 'otro'
       END
FROM prueba;
```

```
a | case
---+-----
1 | uno
2 | dos
3 | otro
4 | otro
```

La cláusula GROUP BY

La cláusula GROUP BY indica que la tabla resultante mostrará los valores agrupados de acuerdo con la lista de atributos que allí se especifiquen y la cláusula HAVING indica que se están filtrando ciertos grupos que no cumplen las condiciones que se establecen para ellos.

Las consultas de grupo producen una fila por grupo. Cuando se efectúan este tipo de consultas, se emplean funciones que permiten resumir la información como son:

- SUM () calcula el total de todas las filas que satisfacen una condición para una columna numérica.
- AVG () calcula el promedio.
- MAX () calcula el valor máximo para una columna.
- MIN () calcula el valor mínimo.
- COUNT(*) calcula el número de filas que satisfacen una condición.

/* En el ejemplo se muestra de los registros académicos: el código del curso, la cantidad de estudiantes y la nota promedio redondeada a un decimal por cada curso */

```
SELECT id_cur, count(*), round(avg(nota), 1)
FROM registro
GROUP BY id_cur;
```

/* La siguiente consulta muestra lo mismo que la anterior, pero se usa la cláusula HAVING para mostrar aquellas donde los grupos, en promedio, han perdido el curso*/

```
SELECT id_cur, count(*), round(avg(nota), 1)
FROM registro
GROUP BY id_cur}
HAVING round(avg(nota), 1) < 3.0;
```

En el próximo ejemplo se está presentando el precio mínimo, el promedio y el máximo por categoría. Se puede decir que agrupe por el número de la columna que en este ejemplo es la 1.

```
SELECT categoriaid, min(preciounit) precio_minimo, round(avg(preciounit), 2) precio_medio,
max(preciounit) precio_max, count(*) cantidad_productos
FROM productos
GROUP BY 1
```

pane					
Output Explain Messages History					
categoriaid	precio_minimo	precio_medio	precio_max	cantidad_pr	
integer	numeric	numeric	numeric	bigint	
200	1.60	2.78	4.30	4	
600	9.80	12.47	14.70	3	
100	2.60	3.32	4.70	5	

Si estas funciones de grupo no se acompañan con la cláusula GROUP BY, entonces se considerará como único grupo a la totalidad de elementos o tuplas que forman la relación que aparece en el FROM.

```
SELECT min(preciounit) precio_minimo, round(avg(preciounit), 2) precio_medio,
       max(preciounit) precio_max, count(*) cantidad_productos
FROM productos
```

ane				
Output Explain Messages History				
	precio_minimo numeric	precio_medio numeric	precio_max numeric	cantidad_productos bigint
	1.60	5.43	14.70	12

La operación JOIN o REUNIÓN

En muchas ocasiones se requiere el enlace o “join” de dos o más tablas. Ese enlace significa juntar las columnas de las dos tablas mediante un atributo en común. Para hacer un enlace, basta con especificar en la cláusula FROM del SELECT, las tablas que se van a enlazar separadas por comas y en la cláusula WHERE colocar los atributos por los cuales se hace el enlace entre las tablas especificadas.

En la siguiente consulta se enlazan las tablas de clientes con la de órdenes de compra para mostrar el nombre de la empresa, el número de la orden y la fecha.

```
SELECT nombre, ordenid, fechaorden
FROM clientes, ordenes
WHERE id = clienteid;
```

Output pane				
Data Output Explain Messages History				
	nombre character(30)	ordenid integer	fechaorden date	
1	SU TIENDA	1	2007-06-17	
2	SU TIENDA	2	2007-06-02	
3	SUPERMERCADO DORADO	3	2007-06-05	
4	MI COMISARIATO	4	2007-06-06	
5	SUPERMERCADO DESCUENTO	5	2007-06-09	
6	SUPERMERCADO DORADO	6	2007-06-12	
7	SUPERMERCADO DORADO	7	2007-06-14	
8	EL ROSADO	8	2007-06-13	
9	EL ROSADO	9	2007-06-17	
10	EL ROSADO	10	2007-06-18	

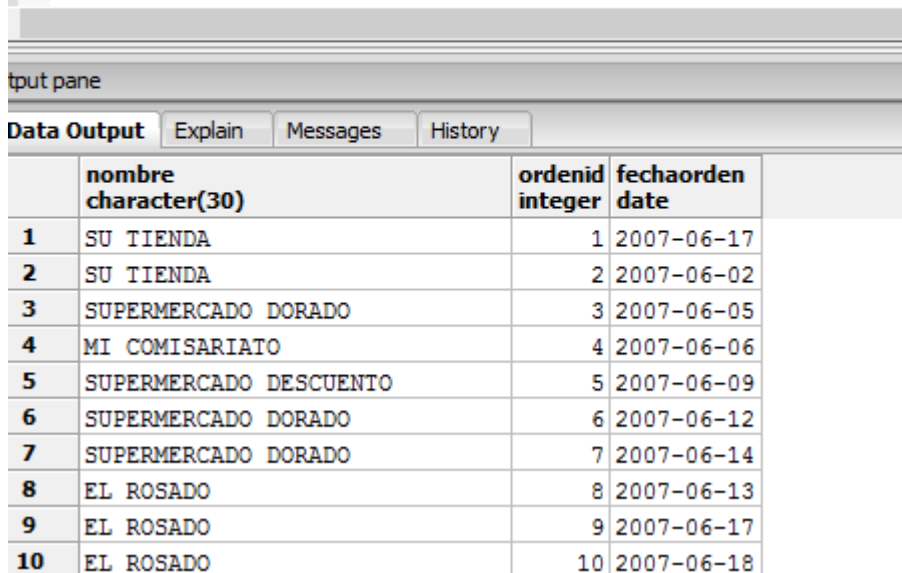
Una forma alternativa para hacer un *join* simple es utilizar explícitamente las palabras INNER JOIN. La sintaxis para ello es:


```
SELECT tabla1.columna1, tabla2.columna2...
FROM tabla1
[INNER] JOIN tabla2
ON tabla1.atributocomún = tabla2. atributocomún;
```

Se debe observar la notación punto (nombretabla.atributo) utilizada para desambiguar porque se permite que un atributo se llame igual en dos o más tablas. Cuando no se llame igual, entonces no es necesario anteponer el nombre de la tabla. También, INNER es opcional en PostgreSQL.

Esta nueva orden equivale a la recién presentada pero usando INNER JOIN

```
SELECT nombre, ordenid, fechaorden
FROM clientes
INNER JOIN ordenes
ON id = clienteid;
```



The screenshot shows a database interface with a query window and a results window. The query window contains the SQL query: `SELECT nombre, ordenid, fechaorden FROM clientes INNER JOIN ordenes ON id = clienteid;`. The results window, titled 'Data Output', shows a table with 10 rows. The columns are 'nombre', 'ordenid', and 'fechaorden'. The data is as follows:

	nombre character(30)	ordenid integer	fechaorden date
1	SU TIENDA	1	2007-06-17
2	SU TIENDA	2	2007-06-02
3	SUPERMERCADO DORADO	3	2007-06-05
4	MI COMISARIATO	4	2007-06-06
5	SUPERMERCADO DESCUENTO	5	2007-06-09
6	SUPERMERCADO DORADO	6	2007-06-12
7	SUPERMERCADO DORADO	7	2007-06-14
8	EL ROSADO	8	2007-06-13
9	EL ROSADO	9	2007-06-17
10	EL ROSADO	10	2007-06-18

Así como los atributos pueden llevar alias, que son nombre temporales, para mayor claridad, las tablas también lo pueden llevar pero con el propósito de evitar nombres tan largos en las sentencias, en especial cuando se hacen enlaces en muchas tablas cuyos atributos tienen nombre iguales. Por ejemplo:

```
SELECT e.nombre, nota, c.nombre
FROM estudiante e, registro r, curso c
WHERE e.cc = registro.cc AND
      registro.id_cur = c.id AND
      nota IS NOT NULL
```

Operaciones de reunión externa (OUTER JOIN)

La reunión externa o OUTER JOIN es una extensión del INNER JOIN. El estándar SQL define tres tipos de reuniones: IZQUIERDA, DERECHA y COMPLETA.

Se puede utilizar las operaciones de OUTER JOIN para enlazar los registros de tablas; trayendo, además, las tuplas que tienen valores que no coinciden con el criterio de enlace en la otra tabla.

La reunión externa izquierda (LEFT JOIN)

En el caso de un OUTER LEFT JOIN, primero se realiza la reunión interna o INNER JOIN. Luego, para cada fila en la tabla T1 que no satisface la condición del enlace con cualquier fila en la tabla T2, se agrega una fila rellena con valores nulos en las columnas de T2. Por lo tanto, la tabla resultante siempre tiene al menos una fila por cada fila en T1.

La siguiente es la sintaxis:

```
SELECT lista  
FROM tabla1 LEFT JOIN tabla2 ON expresión condicional
```

En el ejemplo siguiente vemos los clientes, junto con sus órdenes y también figuran aquellos que aún no han comprado nada:

```
SELECT nombre, ordenid, fechaorden
FROM clientes
LEFT JOIN ordenes
ON id = clienteid;
```

pane

Data Output Explain Messages History

	nombre character(30)	ordenid integer	fechaorden date
	SU TIENDA	1	2007-06-17
	SU TIENDA	2	2007-06-02
	SUPERMERCADO DORADO	3	2007-06-05
	MI COMISARIATO	4	2007-06-06
	SUPERMERCADO DESCUENTO	5	2007-06-09
	SUPERMERCADO DORADO	6	2007-06-12
	SUPERMERCADO DORADO	7	2007-06-14
	EL ROSADO	8	2007-06-13
	EL ROSADO	9	2007-06-17
	EL ROSADO	10	2007-06-18
	YARBANTRELLA		
	EL DESCUENTO		
	SUPERMERCADO ESTRELLA		
	DISTRIBUIDORA PRENSA		
	DE LUISE		

Para saber, por ejemplo, cuáles clientes no han comprado nada todavía, la sentencia sería como se muestra en la página siguiente.

```
SELECT nombre
FROM clientes
LEFT JOIN ordenes
ON id = clienteid
WHERE ordenid IS NULL
```

put pane

Data Output Explain Messages History

	nombre character(30)
1	YARBANTRELLA
2	EL DESCUENTO
3	SUPERMERCADO ESTRELLA
4	DISTRIBUIDORA PRENSA
5	DE LUISE

La reunión externa derecha (RIGHT JOIN)

Primero, se realiza una reunión interna. Luego, para cada fila en la tabla T2 que no cumple la

condición del join con cualquier fila en la tabla T1, se agrega una fila unida con valores nulos en las columnas de T1. Esto es lo contrario de una reunión izquierda; La tabla de resultados siempre tendrá una fila para cada fila en T2.

```
SELECT descripcion, nombrecat
FROM productos
RIGHT JOIN categorias
ON productos.categoriaid = categorias.categoriaid;
```

Output pane	
Output	Explain
Messages	History
descripcion character(50)	nombrecat character(50)
1	SALCHICHAS VIENESAS
2	SALAMI DE AJO
3	BOTON PARA ASADO
4	SALCHICHAS DE POLLO
5	JAMON DE POLLO
6	YOGURT NATURAL
7	LECHE CHOCOLATE
8	YOGURT DE SABORES
9	CREMA DE LECHE
0	BASE DE MAQUILLAJE
1	RIMMEL
2	SOMBRA DE OJOS
3	
4	
5	
6	

La reunión externa completa (FULL OUTER JOIN)

Primero, se realiza una unión interna. Luego, para cada fila en la tabla T1 que no satisface la condición de unión con cualquier fila en la tabla T2, se agrega una fila unida con valores nulos en las columnas de T2. Además, para cada fila de T2 que no cumple la condición de unión con ninguna fila en T1, se agrega una fila unida con valores nulos en las columnas de T1.

La siguiente es la sintaxis:

```
SELECT lista
FROM tabla1 FULL OUTER JOIN tabla2 ON expresión condicional
```

Por ejemplo:

```
SELECT nombre, ordenid
FROM clientes
FULL OUTER JOIN ordenes
ON id = clienteid
```

Otra forma de especificar un enlace, en algunos sistemas gestores de bases de datos como Oracle, es agregar el símbolo "+" encerrado entre paréntesis, en la cláusula WHERE en el lado sobre el cual se quiera hacer el JOIN externo.

Sintaxis

```
SELECT lista
FROM tabla1, tabla2
WHERE tabla1.coumna1[(+)] = tabla2.columna2 [(+)]
```

El siguiente ejemplo se muestra cómo hacer un RIGHT JOIN entre las tablas de órdenes de compra y productos.

```
SELECT nombre, ordenid
FROM clientes, ordenes
WHERE id = clienteid (+)
```

SUBCONSULTAS : (Subquerys).

Una subconsulta es una instrucción SELECT anidada dentro de una instrucción SELECT, INSERT, DELETE, o UPDATE, o bien dentro de otra subconsulta. En muchos casos, puede resultar más rápido y sencillo utilizar una subconsulta que una operación JOIN.

Sintaxis

La sintaxis necesaria para crear una subconsulta tiene tres formatos diferentes que se muestran a continuación.

1. Comparación [ANY | ALL] (instrucciónSelect)
2. Expresión [NOT] IN (instrucciónSelect)
3. [NOT] EXISTS (instrucciónSelect)

En esos formatos la *instrucciónSelect* sigue las mismas reglas que cualquier otra consulta. Debe ir entre paréntesis para que se ejecute primero y el valor traído en la subconsulta deber ser del mismo dominio del valor con el que se va a comparar.

En la lista de atributos de una instrucción SELECT y en las cláusulas WHERE y HAVING se puede especificar una subconsulta en lugar de una expresión. En las subconsultas, la instrucción SELECT proporciona un conjunto de uno o más valores determinados que evaluar en las expresiones de las cláusulas WHERE y HAVING.

Se utilizan los predicados ANY o SOME, que son sinónimos, para obtener en la consulta principal registros que satisfagan la comparación con algún registro obtenido en la subconsulta. Para su correcto funcionamiento, la subconsulta debe devolver exactamente una columna. El predicado ANY debe ir precedido por uno de los siguientes operadores de comparación =, <=, >, <,> y <>. Devuelve verdadero si algún valor de la subconsulta cumple la condición; de lo contrario, devuelve falso.

En el ejemplo siguiente, se obtienen los productos y su precio, que están incluidos en las ventas con un descuento mayor al 8%:

```
SELECT descripcion, preciounit FROM productos
WHERE productoid = ANY
(SELECT productoid FROM detalle_ordenes, ordenes
WHERE ordenes.ordenid = detalle_ordenes.ordenid AND
descuento > 8)
```

pane	
a Output Explain Messages History	
descripcion character(50)	preciounit numeric
SALAMI DE AJO	3.60
BOTON PARA ASADO	4.70
JAMON DE POLLO	2.80
LECHE CHOCOLATE	1.60
BASE DE MAQUILLAJE	14.70
RIMMEL	12.90
SOMBRA DE OJOS	9.80

Se usa el predicado ALL para obtener sólo los registros de la consulta principal que satisfagan la comparación con todos los registros obtenidos en la subconsulta. Esta condición es mucho más restrictiva.

El predicado IN sirve para obtener sólo aquellos registros de la consulta principal para los que algún registro de la subconsulta contenga un valor igual. En el ejemplo siguiente se obtienen todos los productos vendidos con un 10% de descuento o más:

```

SELECT * FROM Productos
    WHERE productoid IN
        (SELECT productoid FROM detalle_ordenes, ordenes
        WHERE ordenes.ordenid = detalle_ordenes.ordenid AND
        descuento > 8)

```

pane

a Output Explain Messages History

	productoid integer	proveedorid integer	categoriaid integer	descripcion character(50)	preciounit numeric	existencia integer
	2	10	100	SALAMI DE AJO	3.60	300
	3	10	100	BOTON PARA ASADO	4.70	400
	5	20	100	JAMON DE POLLO	2.80	100
	7	30	200	LECHE CHOCOLATE	1.60	90
	10	50	600	BASE DE MAQUILLAJE	14.70	40
	11	50	600	RIMMEL	12.90	20
	13	60	600	SOMBRA DE OJOS	9.80	100

A la inversa, se puede utilizar NOT IN para obtener sólo aquellos registros de la consulta principal para los que ningún registro de la subconsulta contiene un valor igual. Debe notarse que el predicado “= ANY” es igual al predicado “IN”. Sin embargo, “<> ANY” no equivale a “NOT IN”.

El predicado EXISTS (con la palabra reservada opcional NOT) en comparaciones con resultado verdadero o falso, sirve para saber si la subconsulta devuelve algún registro. Por otro lado, se puede utilizar en una subconsulta, un alias de nombres de tablas para hacer referencia a tablas que aparezcan en una cláusula FROM afuera de la subconsulta. En el siguiente ejemplo se da a la tabla productos el alias "p". Se hace puesto que se quiere saber cuáles productos tienen un precio mayor al precio promedio de su categoría.

```

SELECT descripcion, preciounit, categoriaid_
FROM productos p
    WHERE p.preciounit >
        (SELECT avg(preciounit) FROM productos
        WHERE p.categoriaid = categoriaid)

```

pane

a Output Explain Messages History

	descripcion character(50)	preciounit numeric	categoriaid integer
	SALAMI DE AJO	3.60	100
	BOTON PARA ASADO	4.70	100
	YOGURT NATURAL	4.30	200
	CREMA DE LECHE	3.60	200
	BASE DE MAQUILLAJE	14.70	600
	RIMMEL	12.90	600

Otros ejemplos de subconsultas de SQL

/* Se obtiene una lista con el nombre, cargo y salario de todos los agentes de ventas cuyo salario es mayor que el de todos los jefes y directores.*/

```
SELECT Apellidos, Nombre, Cargo, Salario
FROM Empleados
WHERE Cargo LIKE "Agente Ven%" AND Salario > ALL
      (SELECT Salario FROM Empleados WHERE (Cargo LIKE
      "%Jefe%" ) OR (Cargo LIKE "%Director%"));
```

/* Se obtiene una lista con el nombre y el precio unitario de todos los productos con menor precio o igual al yogurt */.

```
SELECT descripcion, p.preciounit
FROM Productos p WHERE p.Preciounit = ANY
      (SELECT preciounit FROM Productos
      WHERE descripcion like 'YOGURT%' AND
      p.descripcion not like 'YOGURT%')
```

/* Se obtiene una lista de las compañías que aún no han hecho ninguna compra

```
SELECT
      nombrecia,
      nombrecontacto
FROM
      clientes where clienteid
      NOT IN (SELECT DISTINCT clienteid FROM ordenes)
```

OPERACIONES DE CONJUNTOS

El lenguaje SQL también admite las operaciones de conjunto: La unión, la intersección y la diferencia de conjuntos que se forman a partir de declaraciones SELECT. Para ello, las tablas resultantes de las consultas deben ser compatibles. Esto es, deben tener la misma cantidad de columnas o atributos y los mismos dominios. Además, los atributos deben tener el mismo nombre y por eso, es frecuente el uso de alias para los nombres de las columnas. El resultado de estas operaciones es una sola relación o tabla.

La unión

El operador UNION junta los conjuntos de resultados de dos o más declaraciones SELECT en un único conjunto de resultados. Como ejemplo, la siguiente consulta de unión consiste en dos instrucciones SELECT que devuelven los nombres de la compañía y las ciudades que se encuentran tanto en la tabla Proveedores como en la tabla Clientes pero que sean de Brasil.

```
SELECT Compañia, Ciudad
```



```
FROM Proveedores
WHERE País="Brasil"
UNION
SELECT Compañía, Ciudad
FROM Clientes
WHERE País="Brasil";
```

El operador UNION elimina todas las filas duplicadas a menos que se use UNION ALL.

La intersección

El operador INTERSECT devuelve las filas que están en ambas consultas. Por ejemplo, la siguiente orden trae aquellas personas que al tiempo son profesores y estudiantes:

```
SELECT cc, nombre
FROM profesor
INTERSECT
SELECT cc, nombre
FROM estudiante;
```

La diferencia de conjuntos en SQL

El operador EXCEPT devuelve filas de la primera consulta que no están en el resultado de la segunda consulta. . Por ejemplo, la siguiente orden trae aquellas personas que son profesores pero no estudiantes:

```
SELECT cc, nombre
FROM profesor
EXCEPT
SELECT cc, nombre
FROM estudiante;
```

Consultando el diccionario de datos

Del mismo modo cómo se consultan los datos en la base de datos con una sentencia SELECT, así mismo se puede consultar el catálogo de datos, también llamado diccionario de datos que contiene todas las descripciones de los objetos creados y sus características. Por ejemplo, la siguiente sentencia, muestra las tablas creadas en el esquema público de PostgreSQL.

```
SELECT * FROM information_schema.tables
WHERE table_schema = 'public';
```

La siguiente orden, muestra los nombres de las columnas de la tabla clientes, el tipo de datos y la cantidad máxima de caracteres, si la columna es de tipo cadena de caracteres:

```
SELECT column_name, data_type, character_maximum_length
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'clientes';
```

Los nombres y objetos creados en el diccionario de datos de cada sistema gestor de bases de datos varían pues no existe un estándar. En Oracle, se utilizan las instrucciones siguientes, entre otras:

DESC USER_TABLES	Para ver las tablas del usuario activo
SELECT TABLE_NAME FROM USER_TABLES;	Para saber las tablas que tiene el usuario
USER_CONS_COLUMNS DBA_CONS_COLUMNS ALL_CONS_COLUMNS	Para ver las restricciones que afectan a las columnas. Del usuario, definidas por el DBA y por todas las personas.

Manejo de Vistas

La sintaxis para crear vistas es:

```
CREATE OR REPLACE VIEW <nombre_vista> [(nombrecol1, nombrecol2...)]
AS
  SELECT ...
[WITH (check_option=cascaded|local)];
```

Por ejemplo, para crear una vista de los productos cárnicos, se haría así:

```
CREATE OR REPLACE VIEW prod_carnicos AS
  SELECT productoid, proveedorid, descripcion, preciounit,
         existencia
  FROM productos
  WHERE productos.categoriaid = 100;
```

En el ejemplo anterior, así como en la creación de tablas, también se puede utilizar la palabra REPLACE para reemplazar una vista preexistente, con el mismo nombre. Otros ejemplos son:

```
/* Esta vista es una proyección del álgebra relacional. Impide ver el sueldo de los empleados*/
```

```
CREATE VIEW empleado
AS SELECT nombre, cargo, fecha_ing FROM emp;

CREATE VIEW Vuelos_Medellin_
(procedencia, destino, duracion)
AS SELECT origen, destino, fecha_ini-fecha_fin
FROM vuelos
WHERE ORIGEN = 'Medellín'
```

Para crear vistas actualizables, se deben cumplir los siguientes requisitos:

- La consulta de definición de la vista debe tener exactamente una entrada en la cláusula FROM, que puede ser una tabla u otra vista actualizable.
- La consulta de definición no debe contener una de las siguientes cláusulas: GROUP BY, HAVING, LIMIT, DISTINCT, UNION, INTERSECT y EXCEPT.
- La lista de selección no debe ninguna función agregada como SUM, COUNT, AVG, MIN y MAX.

En una vista actualizable se puede usar la cláusula WITH CHECK OPTION para verificar la condición que define la vista cuando realiza un cambio en la tabla base a través de la vista.

El ejemplo siguiente, crea una vista actualizable de las ciudades de Colombia:

```
CREATE VIEW ciudades_col AS SELECT
ciudad, id_pais
FROM
ciudad
WHERE id_pais = 57
WITH CHECK OPTION;
```

Hacer uso de las vistas es un aspecto clave de un buen diseño de base de datos SQL. Las vistas le permiten encapsular los detalles de la estructura de las tablas, que pueden cambiar en el tiempo, detrás de interfaces consistentes. Las vistas se pueden utilizar en casi cualquier lugar donde se pueda usar una tabla real. Construir vistas sobre otras vistas, tampoco es raro. En el ejemplo siguiente se muestra cómo listar únicamente las ciudades de Colombia o los productos cárnicos, de una manera más sencilla, a través de una vista.

```
SELECT * FROM ciudades_col;
SELECT * FROM prod_cárnicos;
```

Como estas dos vistas son actualizables, también se puede dar una orden de inserción, borrado o actualización. En el ejemplo, se ingresa una nueva ciudad colombiana:

```
INSERT INTO ciudades_col (ciudad, idpais)
VALUES ('San José', 57);
```

Si se intenta ingresar una ciudad de otro país a través de la vista, no se podría porque la vista está definida para las ciudades de Colombia únicamente y se puso la opción de chequeo.

Para cambiar la definición de una vista, se usa la declaración ALTER VIEW, con la siguiente sintaxis:

```
ALTER VIEW [IF EXISTS] nombre ALTER [COLUMNA] nombre_col SET  
    DEFAULT expression;
```

```
ALTER VIEW [IF EXISTS] nombre ALTER [COLUMNA] nombre_col DROP  
    DEFAULT;
```

```
ALTER VIEW [IF EXISTS] nombre OWNER TO  
    {Nuevo_usuario |USER (usuario actual)};
```

```
ALTER VIEW [IF EXISTS] nombre RENAME TO nuevo_nombre;
```

La sentencia ALTER VIEW cambia varias propiedades auxiliares de una vista. Si desea modificar la consulta de definición de la vista, se debe usar la sentencia CREATE OR REPLACE VIEW.

Si se desea eliminar una vista se usa la sentencia DROP. Por ejemplo:

```
DROP VIEW prod_carnicos;
```

Manejo de Índices

Un índice, como en los libros convencionales, es una manera más rápida de llegar a los datos requeridos porque cuando se crea un índice, se guardan los valores ordenados de las columnas por las cuales se indexó junto con el puntero o dirección física de cada registro o tupla. Como ejemplo, vamos a emplear la tabla "libros" cuya definición es la siguiente:

```
CREATE TABLE libros(  
    codigo int not null,  
    titulo varchar(40) not null,  
    autor varchar(30),  
    editorial varchar(15),  
    precio decimal(6,2)  
);
```

Puesto que un atributo por el cual se realizan consultas frecuentemente es "autor", sería conveniente indexar la tabla por ese atributo. El índice se crearía así:

```
CREATE INDEX I_libros_autor ON libros(autor);
```

Cuando un índice se declara único, no se permiten varias filas de tablas con valores indexados iguales. Los valores nulos no se consideran iguales. Un índice único de varias

columnas solo rechazará los casos en los que todas las columnas indexadas en combinación sean iguales en varias filas.

Un índice único se crea automáticamente cuando se define una restricción única (UNIQUE) o clave primaria (PRIMARY KEY) para una tabla. El índice incluye las columnas que conforman la clave principal o la restricción única (puede ser un índice de varias columnas).

Manejo de Disparadores

Un disparador, como se dijo antes, es una función que se invoca automáticamente cada vez que ocurre un evento asociado con una tabla. Un evento podría ser: INSERTAR, ACTUALIZAR o BORRAR.

Para crear un nuevo disparador, primero debe definir una función de activación y luego vincular esta función de activación a una tabla. La diferencia entre un disparador y una función definida por el usuario es que un disparador se invoca automáticamente cuando ocurre el evento.

Se proporcionan dos tipos principales de disparadores: desencadenantes de fila y de sentencia. Las diferencias entre los dos son cuántas veces se activa el disparador y en qué momento. Por ejemplo, si emite una sentencia UPDATE que afecta a 20 filas, el disparador de fila se invocará 20 veces, mientras que el disparador de instrucción se invocará una sola vez.

Se puede especificar si el disparador se invoca antes o después de un evento. Si el activador se invoca antes de un evento, se puede omitir la operación de la fila actual o incluso cambiar la fila que se está actualizando o insertando. En caso de que se invoque el disparador después del evento, todos los cambios están disponibles para el disparador.

La sintaxis para crear un disparador en PostgreSQL es:

```
CREATE TRIGGER nombre {BEFORE | AFTER | INSTEAD OF} {event [OR ...]}  
    ON tabla  
    [FOR [EACH] {ROW | DECLARACIÓN}]  
    EXECUTE PROCEDURE función
```

Suponga que se quiere tener un histórico de los cambios de sueldo por las promociones que tienen los profesores, entonces se definirá un disparador así:

```
CREATE TRIGGER cambio_por_promoción  
    BEFORE UPDATE  
    ON profesores  
    FOR EACH ROW  
    EXECUTE PROCEDURE nuevo_reg_promocion();
```

Donde el procedimiento “nuevo_reg_promocion”, en PG/PLSQL que es el lenguaje procedimental en PostgreSQL, es:

```
CREATE OR REPLACE FUNCTION nuevo_reg_promocion()  
  RETURNS trigger AS  
$BODY$  
BEGIN  
IF NEW.salario <> OLD.salario THEN  
INSERT INTO auditoria_prom(id,nombre,salario_ant, fecha)  
      VALUES (OLD.id,OLD.nombre,OLD.salario, now());  
END IF;  
RETURN NEW;  
END;  
$BODY$
```

En Oracle, la sintaxis de un disparador es ligeramente diferente. El siguiente es un ejemplo de un disparador que se activará cuando se cambie el código de un departamento, para cambiar el código del departamento de la tabla de empleados por ese mismo nuevo valor a todos los empleados que pertenezcan al mismo. Esto se conoce como una actualización en cascada:

```
CREATE TRIGGER actualizacion_cascada  
BEFORE UPDATE OF id ON depto  
FOR EACH ROW  
BEGIN  
UPDATE emp  
SET iddepto =:new.id  
WHERE iddepto =:old.id;  
END;
```

Debe observarse que la diferencia entre la forma de especificar los disparadores no es mucha. En Oracle, los valores anteriores y posteriores de una transacción se guardan en variables y se distinguen por los dos puntos que se utilizan como prefijos. En PostgreSQL, no se utilizan.

Para eliminar una vista se usa la sentencia DROP, con la siguiente sintaxis:

```
DROP VIEW [IF EXISTS] nombre [,...] [ CASCADE | RESTRICT ]
```

Si se incluye la palabra opcional CASCADE se eliminan todos los objetos que dependan de esa vista. En cambio RESTRICT impide borrar la vista si hay otros objetos que dependen de ella. Esta última opción es la que existe por defecto.

Manejo de transacciones

El SQL es un lenguaje que también permite el control de las transacciones. Una transacción es un conjunto de operaciones que se deben realizar como un todo y no parcialmente porque puede llevar la base de datos a un estado inconsistente.

En PostgreSQL, una transacción se configura rodeando los comandos SQL de la transacción con los comandos `BEGIN` (inicio) y `COMMIT` (confirmar). Entonces nuestra transacción bancaria se vería como esta:

```
BEGIN;
UPDATE cuentas SET saldo = saldo - 100.00
WHERE cc = '8145589';
...
COMMIT;
```

Si, en la mitad de la transacción, por alguna razón no se desea confirmar la transacción (porque el saldo daría negativo), podemos emitir el comando `ROLLBACK` en lugar de `COMMIT`, y todas las actualizaciones hasta el momento serían canceladas.

PostgreSQL en realidad trata cada sentencia SQL como ejecutada dentro de una transacción. Si no se emite un comando `BEGIN`, entonces cada declaración individual tiene un `BEGIN` implícito y (si es exitoso) un `COMMIT`. Un grupo de sentencias entre el `BEGIN` y el `COMMIT` a veces se denomina bloque de transacción.

Es posible controlar las declaraciones en una transacción de forma más granular mediante el uso de puntos de grabación. Permiten descartar de forma selectiva partes de la transacción, mientras que el resto se confirma. Después de definir un punto de grabación con `SAVEPOINT`, puede, si es necesario, volver a ese punto de con `ROLLBACK TO`. Todos los cambios en la base de datos de la transacción entre la definición del punto de salvaguarda y la reversión a ella se descartan, pero los cambios anteriores si se guardan. En el ejemplo siguiente, se devuelve la transacción hasta el “punto_a”, si se cumple cierta condición.

```
BEGIN;
UPDATE cuentas SET saldo = saldo - 100.00
WHERE cc = '815550';
SAVEPOINT punto_a;
UPDATE accounts SET balance = balance + 100.00
    WHERE cc = '15489';
...
IF... THEN ROLLBACK TO punto_a;
COMMIT;
```

Hasta este punto se han descrito las principales sentencias que se pueden especificar en el lenguaje SQL, con ejemplos de las distintas declaraciones, pertenecientes tanto al sublenguaje DML como al DDL. El manejo de transacciones hace parte del DCL o lenguaje de control.

En el próximo capítulo se describirán las interfaces del gestor de bases de datos PostgreSQL y del sistema Hive del entorno Hadoop para bases de datos masivas. Ambos gestores son de dominio público e incluso este último entorno puede accederse en la nube.