1. SQL

SQL es un lenguaje estructurado de consulta (Structured Query Language, en inglés). diseñado específicamente para administrar información en sistemas de gestión de bases de datos de tipo relacional.

Puede considerarse un lenguaje de programación como tal, ya que cuenta con uso de variables, tipos de datos, elementos condicionales y lógicos. Es el estandar de facto para la gestión de datos y permite:

- Consultar, actualizar y reorganizar datos
- Crear y modificar la estructura de datos
- Controlar el acceso a los datos

El uso de este lenguaje es altamente imperativo para todo profesional que tenga por objetivo acceder a altos volúmenes y/o datos complejos.

El núcleo central de SQL se utiliza en la gran mayoría de las bases de datos comerciales y de uso gratuito. Algunas compañías han realizado ajustes a SQL para adaptarlas de mejor manera a sus productos y originado por tanto versiones "mejoradas" de SQL, como ejemplo de ello PL/SQL de Oracle y Transact-SQL de Microsoft.

SQL es estándar ya que se utiliza en la totalidad de las bases conocidas, ya sea en forma nativa o en alguna variante según el fabricante de la base de datos.

Un ejemplo de SQL en el siguiente bloque de código:

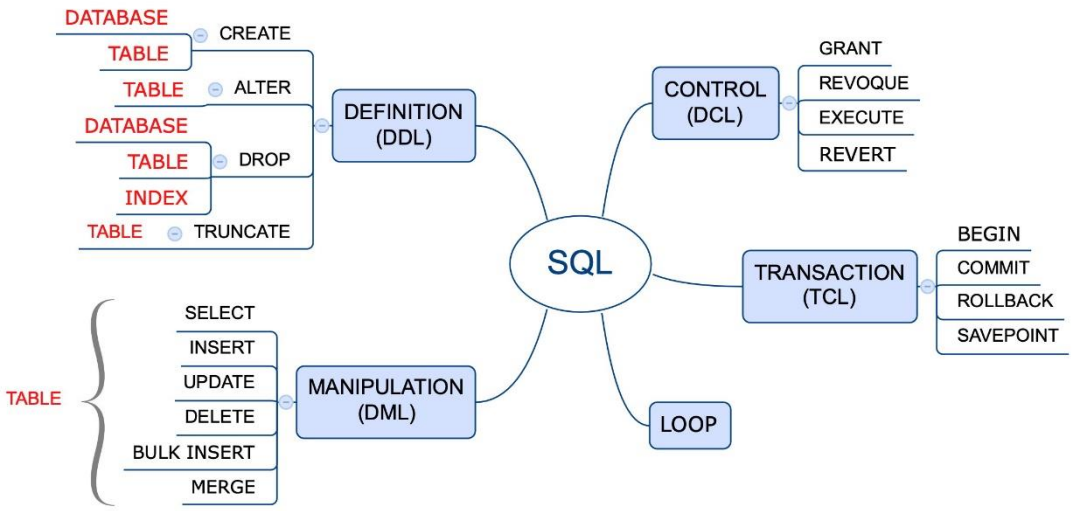
```
SELECT nombre, apellido, rut, edad, genero

FROM clientes

WHERE edad > 20 AND genero = 'M'

ORDER BY edad ASC
```

Donde: **SELECT** permite elegir que campos (o columnas como nombre, apellido, rut, edad y genero) queremos mostrar, **FROM** nos indica desde que tabla (clientes) y donde están definidos los campos. **WHERE** permite especificar una condición (o filtro que queremos ejecutar sobre los datos resultantes) y **ORDER BY** nos permite obtener una lista ordenada por edad en forma ascendente.

Por tanto, el código anterior obtendrá una lista de los clientes mayores de 20 años de género masculino y entregará los resultados en forma ordenada de menor a mayor edad. La lista contendrá las variables de nombre, apellido, rut, edad y género en ese mismo orden.

SQL posee entornos específicos de uso. Posee funciones para:

- La **definición de las bases de datos** (DDL),
- El **control de acceso de usuarios** (DCL),
- La **gestión de las transacciones** (TCL)
- Y el más importante o más utilizados la **manipulación de datos** (DML) que posee las funciones para la obtención de los datos.



1.1 Elementos Bases de datos

A parte de las tablas como ya hemos visto existen otros tipos de objetos en las bases de datos que podremos usar como los índices y las vistas.

Índice

El índice de una base de datos es una estructura de datos que mejora la velocidad de las operaciones, permitiendo un rápido acceso a los registros de una tabla en una base de datos. Al aumentar drásticamente la velocidad de acceso, se suelen usar sobre aquellos campos sobre los cuales se hacen frecuentes búsquedas.

¿Para qué se utilizan los índices en DB? El índice tiene un funcionamiento similar al índice de un libro, guardando parejas de elementos: el elemento que se desea indexar y su posición en la base de datos. Para buscar un elemento que esté indexado, sólo hay que buscar en el índice dicho elemento para, una vez encontrado, devolver el registro que se encuentre en la posición marcada por el índice. Los índices pueden ser creados usando una o más columnas, proporcionando la base tanto para búsquedas rápidas al azar como de un ordenado acceso a registros eficiente.

Vistas

Una vista de base de datos es un subconjunto de una base de datos y se basa en una consulta que se ejecuta en una o más tablas de base de datos. Las vistas de base de datos se guardan en la base de datos como consultas con nombre y se pueden utilizar para guardar consultas completas que se utilizan con frecuencia.

1.2 Tipos de Datos

Los tipos de datos soportados lo son por la mayoría de los sistemas de bases de datos que soportan SQL, salvo algunas excepciones que son particulares a un sistema específico.

En general los tipos de datos manejados por SQL incluyen si bien luego cada base de datos tiene sus propios tipos:

- Numéricos
- Moneda
- Carácter
- Binario
- Fecha/hora
- Lógicos (booleanos)
- Enumerados
- Geométricos
- Redes
- Bit String
- Texto
- UUID
- XML
- JSON
- Arreglos
- Compuestos (Composite)
- Rangos
- Identificadores de objetos

1.3 Definición de Datos (DDL)

Este lenguaje está orientado a la definición de las estructuras de datos dentro de la base de datos. Las estructuras son las tablas, índices, vistas, etc. Las funciones principales son CREATE, ALTER y DROP (crear, modificar y eliminar respectivamente).

1.3.1 CREATE TABLES

```
-- crea una base de datos

CREATE DATABASE midb;

USE midb;

-- crea las tablas dentro de la base de datos midb

CREATE TABLE mitabla ( id INT PRIMARY KEY, nombre VARCHAR(20) );

INSERT INTO mitabla VALUES ( 1, 'Sara' );

INSERT INTO mitabla VALUES ( 2, 'Sonia' );
```

```
INSERT INTO mitabla VALUES ( 3, 'Daniel' );

-- crea una tabla persona

CREATE TABLE person (

person_id BIGINT NOT NULL,

last_name VARCHAR(255) NOT NULL,

first_name VARCHAR(255),

age INT NOT NULL,

PRIMARY KEY (person_id)

);


-- alternativamente define la clave primaria en forma directa

CREATE TABLE person (

person_id BIGINT NOT NULL PRIMARY KEY,

last_name VARCHAR(255) NOT NULL,

first_name VARCHAR(255),

address VARCHAR(255),

city VARCHAR(255)

);
```

- `NOT NULL` - este parámetro asegurará que la columna no almacene un valor `NULL`
- `UNIQUE` - este parámetro va a prevenir que la columna almacene el mismo valor más de una vez.
- `UNIQUE KEY` - este parámetro designará la columna como un identificador único.

Crear una tabla a partir de otra existente.

```
-- crea una nueva tabla a partir de los datos de "person" con registros

-- de personas que tengan sobre 30 años

CREATE TABLE people_over_30 AS SELECT * FROM person WHERE age > 30;
```

1.3.2 CREATE INDICES

```
-- crea un índice en la tabla TB_FILMS para el campo title

CREATE UNIQUE INDEX title_idx ON TB_FILMS (title);
```

1.3.3 CREATE USERS/ROLES

-- crea un usuario/rol en la base de datos

CREATE USER davide **WITH PASSWORD** 'jw8s0F4';

-- crea un rol con una password valida hasta una fecha específica. Luego de esa fecha la password no tendrá validez.

CREATE ROLE miriam **WITH** LOGIN **PASSWORD** 'jw8s0F4' VALID **UNTIL** '2005-01-01';

-- crea un rol que puede crear bases de datos y administrar roles

CREATE ROLE admin WITH CREATEDB CREATEROLE;

1.3.4 CREATE VISTAS

-- crea una vista clientes_ventas

CREATE VIEW clientes_ventas **AS**

SELECT * **FROM** clientes

WHERE region = "RM";

1.3.5 ALTER

Modifica la estructura de una tabla sin suprimirla y volver a crearla, como añadir columnas, eliminar columnas o cambiar definiciones de columna (por ejemplo, longitud o los valores predeterminados).

-- forma general

ALTER TABLE [**IF EXISTS**] [**ONLY**] name [*] action [, ...]

ALTER TABLE [**IF EXISTS**] [**ONLY**] name [*] **RENAME** [**COLUMN**] column_name **TO** new_column_name

ALTER TABLE [**IF EXISTS**] [**ONLY**] name [*] **RENAME CONSTRAINT** constraint_name **TO** new_constraint_name

ALTER TABLE [IF EXISTS] name **RENAME TO** new_name ALTER TABLE [IF EXISTS] name **SET SCHEMA** new_schema

-- agrega una columna direccion a la tabla clientes, como resultado todos los registros tendran ese campo nulo

ALTER TABLE clientes **ADD COLUMN** direccion varchar(30);

```
-- todos los registros existentes recibirán la fecha actual en el nuevo campo fecha
-- los nuevos registros tendrán la fecha en que se ingresen
ALTER TABLE clientes
ADD COLUMN fecha timestamp with time zone DEFAULT now();

-- agregar una columna con un valor inicial que luego cambiará a otro valor
-- para los nuevos registros
ALTER TABLE ventas
ADD COLUMN status varchar(30) DEFAULT 'old',
ALTER COLUMN status SET default 'current';
-- los registros previamente existentes recibirán el valor 'old', en cambio los nuevos tendran el valor 'current'

-- eliminar una columna de una tabla
ALTER TABLE distributors DROP COLUMN address RESTRICT;

-- realizar cambio en dos columnas a la vez
ALTER TABLE clientes
   ALTER COLUMN direccion TYPE varchar(80),
   ALTER COLUMN nombre TYPE varchar(100);

-- renombrar una columna
ALTER TABLE clientes RENAME COLUMN direccion TO ciudad;

-- renombrar una tabla
ALTER TABLE clientes RENAME TO personas;

-- agregar una restricción a una columna
ALTER TABLE clientes ALTER COLUMN calle SET NOT NULL;
```

-- eliminar una restricción en una columna

ALTER TABLE clientes ALTER COLUMN calle DROP NOT NULL;

-- agregar una clave foránea (foreign key) a una tabla

ALTER TABLE clientes **ADD CONSTRAINT** clidireccion **FOREIGN KEY** (direccion) **REFERENCES** direcciones (direccion);

-- agregar una clave primaria a una tabla

ALTER TABLE clientes **ADD PRIMARY KEY** (cli_id);

1.3.6 DROP

Sirve para eliminar objetos de la base de datos.

-- elimina una tabla personas

DROP TABLE personas;

-- elimina la base de datos

DROP DATABASE midb;

-- elimina una vista llamada clientes_ventas

DROP VIEW clientes_ventas;

1.3.7 TRUNCATE

Elimina los datos de una tabla sin alterar la estructura de la tabla.

-- vacía la tabla clientes

TRUNCATE TABLE clientes;

1.4 Manipulación de Datos (DML)

Un lenguaje de manipulación de datos (Data Manipulation Language, o DML en inglés) es una parte del lenguaje SQL que incluido por el sistema de gestión de base de datos permite a los usuarios llevar a cabo las tareas de consulta o manipulación de los datos, organizados por el modelo de datos adecuado.

DML está orientado por tanto a la interacción con los datos propiamente tales dentro de la base de datos. Esta interacción incluye la selección, inserción, actualización y eliminación de datos (registros dentro de una BD).

1.4.1 SELECT

SELECT permite recuperar registros desde cero o más tablas. WHERE filtra los registros antes del agrupamiento y HAVING filtra los grupos creados por GROUP BY.

SELECT en PostgreSQL

[**WITH** [RECURSIVE] with_query [, ...]]

SELECT [**ALL** | **DISTINCT** [**ON** (expression [, ...])]] * | expression [[**AS**] output_name] [, ...]

[**FROM** from_item [, ...]]

[**WHERE** condition]

[**GROUP BY** expression [, ...]]

[**HAVING** condition [, ...]]

[WINDOW window_name **AS** (window_definition) [, ...]]

[{ UNION | INTERSECT | EXCEPT } [ALL] select]

[**ORDER BY** expression [**ASC** | **DESC** | **USING operator**]

[**NULLS** { FIRST | LAST }] [, ...]] [**LIMIT** { count | **ALL** }]

[OFFSET **start** [**ROW** | **ROWS**]] [FETCH { FIRST | **NEXT** } [count]

{ **ROW** | **ROWS** } **ONLY**]

[**FOR** { **UPDATE** | **SHARE** } [**OF** table_name [, ...]] [**NOWAIT**] [...]]column2, ...

FROM table_name;

SELECT f.title, f.did, d.name, f.date_prod, f.kind

FROM distributors d, films f

WHERE f.did = d.did

```
SELECT kind, sum(len) AS total
   FROM films
   GROUP BY kind
   HAVING sum(len) < interval '5 hours';
```

```
SELECT *  FROM T_CLIENTES;
SELECT *  FROM T_PEDIDOS;
SELECT *  FROM T_PRODUCTOS;
```

1.4.2 INSERT

Realiza operaciones de ingreso de valores dentro de la tabla.

```
INSERT INTO table_name VALUES (value1, value2, value3)
```

```
INSERT INTO table_name (column1, column2, column3,..) VALUES ( value1, value2, value3,..);
```

```
insert into UTILES  (codigo, descripcion, fecalta) values (1,'Lapiz',sysdate);
insert into UTILES  (codigo, descripcion, fecalta) values (2,'Goma',sysdate);
insert into UTILES  (codigo, descripcion, fecalta) values (3,'Sacapuntas',null);
insert into UTILES  values (4,'regla',sysdate);
insert into UTILES  values (5,'escuadra',null);
insert into UTILES  (codigo, descripcion)  values (6,'transportador');
insert into UTILES  (descripcion, codigo)  values ('compas',7);
insert into UTILES  (fecalta, codigo, descripcion ) values (sysdate, 8, 'estuche');
```

1.4.3 UPDATE

La sentencia UPDATE es utilizada para actualizar los datos de una tabla existente dentro de la base de datos.

Se pueda actualizar desde una a varias columnas según se requiera. Si se omite la claúsula WHERE, el comando afecta a todos los registros de la tabla.

UPDATE estudiantes **SET** NAME = 'PRATIK' **WHERE id** = 20;

UPDATE table_name **SET** column1 = value1, column2 = value2 **WHERE** condition;

donde: + table_name: nombre de la tabla + columnN: nombre de la columna + valueN: nuevo valor de la columna + condition: condición requerida para seleccionar las filas que deben ser actualizadas.

UPDATE nombre_tabla

SET nombre_columna = valor

[**WHERE** condición [**AND** condición]]

UPDATE UTILES **SET** descripcion = 'Regla 30 cm' **WHERE** codigo= 4

UPDATE UTILES **SET** fecalta = sysdate;

UPDATE utiles **SET** fecalta = sysdate, descripcion='regla 30 cm.' **WHERE** código = 4;

1.4.4 DELETE

Realiza operaciones de eliminación de registros desde las tablas.

DELETE FROM nombre_tabla

WHERE nombre_columna = valor

DELETE FROM personas

WHERE nombre = 'LUIS' **AND** apellido1 = 'LOPEZ' **AND** apellido2 = 'PEREZ'

DELETE nombre_tabla

[**WHERE** condición [**AND** condición]]

delete UTILES **WHERE** codigo= 7;

delete UTILES;

1.5 Uniones (JOIN)

La herramienta más potente en el modelo relacional. Las sentencias JOIN permiten el uso de datos provenientes de dos o más tablas utilizando como conector entre ellas una columna cómun. Esta columna debe ser del mismo tipo de datos, y generalmente es la clave primaria de ellas (aunque no necesariamente).

1.5.1 LEFT JOIN

LEFT JOIN o LEFT OUTER JOIN obtiene todos loS datos de la primera tabla (o tabla izquierda), y solamente las filas que coinciden desde la segunda tabla (o tabla derecha). en caso de no haber coincidencias desde esta última tabla muestra solo valores nulos.

-- recuperando filas desde una tabla que no poseen correspondencia de filas a otra tabla

-- mantiene solo las filas que no corresponden (anti-join)

SELECT d.*

FROM departamentos d **LEFT OUTER JOIN** empleados e

ON (d.depto = e.depto)

WHERE e.deptno **is null**

-- seleccionar registros de dos tablas y luego correlacionar con una tercera tabla. empleados y departamentos y bono entregado (no todos los empleados reciben bono)

SELECT e.ename, d.loc, eb.bono

FROM empleados e

JOIN departamentos d **ON** (e.deptno=d.deptno)

LEFT JOIN bonos eb **ON** (e.empno=eb.empno)

ORDER BY 2

-- alternativa

SELECT e.ename, d.loc,

(**SELECT** eb.bono **FROM** bonos eb

WHERE eb.empno=e.empno) **as** bono

FROM empleados e, departamentos d

WHERE e.deptno=d.deptno

-- clientes sin ordenes de compra

SELECT distinct a.cliente_id

```
FROM transacciones a
LEFT JOIN clientes b ON a.cliente_id = b.cliente_id
WHERE b.cliente_id IS NULL

-- cuantos pacientes hay de cada comuna
SELECT count(*) cantidad, co_id codigo, co_comuna as comuna
FROM pacientes p
LEFT JOIN comunas c on (p.pa_comuna = c.co_id)
WHERE pa_comuna !='00000'
GROUP BY co_id, pa_comuna
ORDER BY cantidad DESC;

-- cuantas consultas por usuario medico (completo)
SELECT count(ex_idm),md_nombre||' '||md_apellido , ex_idm as medicos
FROM consultas c
LEFT JOIN medicos ON (ex_idm=md_id)
GROUP BY ex_idm, md_nombre||' '||md_apellido
```

1.5.2 INNER JOIN

INNER JOIN o JOIN regresa solamente los registros coincidentes en ambas tablas.

```
-- sentencia general
SELECT *
FROM table_1
INNER JOIN table_2
ON table_1.column_name = table_2.column_name

-- forma simplificada
SELECT * FROM table_1 JOIN table_2
ON table_1.column_name = table_2.column_name
```

-- detalles de orden y producto

```
SELECT a.*,b.*

FROM ventas as a

INNER JOIN

productos as b

ON a.idproducto = b.idproducto
```

-- Encontrar el total de camas (beds) por nacionalidad dentro de los registros de Airbnb

-- en las tablas airbnd_apartments con airbnd_hosts mediante la columna relacionada "host_id"

```
SELECT nationality, SUM(n_beds) AS total_beds_available

FROM airbnb_hosts h

INNER JOIN airbnb_apartments a ON h.host_id = a.host_id GROUP BY nationality

ORDER BY total_beds_available DESC;
```

-- Esta consulta devolverá los datos de la tabla 1 cuyos campos coincidan con los de la tabla2 con una clave y los datos que no estén en la tabla 1 al compararlos con la tabla2 con una condición y una clave

```
select *

from Table1 t1

inner join Table2 t2 on t1.ID_Column = t2.ID_Column

left join Table3 t3 on t1.ID_Column = t3.ID_Column where t2.column_name = column_value

and t3.ID_Column is null order by t1.column_name;
```

-- Esta consulta devolverá los datos de la tabla 1 cuyos campos coincidan con los de la tabla2 con una clave y los datos que no estén en la tabla 1 al compararlos con la tabla2 con una condición y una clave

```
select *

from Table1 t1

inner join Table2 t2 on t1.ID_Column = t2.ID_Column

left join Table3 t3 on t1.ID_Column = t3.ID_Column where t2.column_name = column_value

and t3.ID_Column is null order by t1.column_name;
```

1.5.3 OUTER JOIN

LEFT OUTER JOIN

Devuelve todas las filas de la tabla izquierda, emparejadas con las filas de la tabla derecha en las que se cumplen las condiciones de la cláusula ON. Las filas en las que no se cumple la cláusula ON tienen NULL en todas las columnas de la tabla derecha

--

```
SELECT * FROM table_1 AS t1

LEFT JOIN table_2 AS t2 ON t1.ID_Column = t2.ID_Column
```

RIGHT OUTER JOIN

Devuelve todas las filas de la tabla derecha, emparejadas con las filas de la tabla izquierda en las que se cumplen las condiciones de la cláusula ON. Las filas en las que no se cumple la cláusula ON tienen NULL en todas las columnas de la tabla izquierda.

--

```
SELECT * FROM table_1 AS t1

RIGHT JOIN table_2 AS t2 ON t1.ID_Column = t2.ID_Column
```

FULL OUTER JOIN

Se devuelven todas las filas de ambas tablas, independientemente de que se cumplan las condiciones de la cláusula ON. Las filas que no satisfacen la cláusula ON se devuelven con NULL en todas las columnas de la tabla opuesta (es decir, para una fila de la tabla izquierda, todas las columnas de la tabla derecha contendrán NULL, y viceversa).

--

```
SELECT * FROM table_1 AS t1

FULL JOIN table_2 AS t2 ON t1.ID_Column = t2.ID_Column
```

1.6 Control de Datos (DCL)

Son una colección de comandos destinados al control del acceso a los objetos y estructuras de la base de datos. Los comandos utilizados son GRANT y REVOQUE (otorgar y denegar respectivamente).

Grant define los privilegios de acceso a los diversos objetos dentro de la base de datos. El estándar SQL no permite establecer los privilegios en más de un objeto por comando.

Revoque elimina los permisos otorgados previamente por GRANT.

Los privilegios son:

- SELECT
- INSERT
- UPDATE
- DELETE
- TRUNCATE
- REFERENCES
- TRIGGER
- CREATE
- CONNECT
- TEMPORARY
- EXECUTE
- USAGE

1.6.1 Ejemplos

-- forma general de GRANT

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }

[, ...] | **ALL** [**PRIVILEGES**] }

ON { [**TABLE**] table_name [, ...]

| **ALL TABLES IN SCHEMA** schema_name [, ...] }

TO role_specification [, ...] [**WITH GRANT OPTION**]

[GRANTED **BY** role_specification]

-- otorga privilegios de ingreso a todos los usuarios para la tabla films:

GRANT INSERT ON films TO PUBLIC;

-- otorga todos los privilegios al usuario manuel en la tabla clientes

GRANT ALL PRIVILEGES ON clientes TO manuel;

-- permite la membresia en el grupo admins al usuario manuel

GRANT admins **TO** manuel;

-- forma general de REVOQUE

REVOKE [GRANT OPTION FOR]

{ { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }

[, ...] | **ALL** [**PRIVILEGES**] }

ON { [**TABLE**] table_name [, ...]

| **ALL TABLES IN SCHEMA** schema_name [, ...] }

FROM role_specification [, ...]

[GRANTED **BY** role_specification]

[CASCADE | RESTRICT]

-- revoca el privilegio de insertar para el el usuario public en l atabla films

REVOKE INSERT ON films FROM PUBLIC;

-- revoca todos los privilegios al usuario manuel para la tabla clientes

REVOKE ALL PRIVILEGES ON clientes FROM manuel;

-- revoca la membresía del grupo admins al usuario manuel

REVOKE admins **FROM** manuel;

1.7 Control de Transacciones (TCL)

Una transacción es una agrupación de acciones que se comportan como una unidad. Estas deben cumplir 4 propiedades fundamentales conocidas como ACID (atomicidad, coherencia, asilamiento y durabilidad).

- **Atomicidad**: Una transacción es indivisible, o se ejecutan todas las sentencias o no se ejecuta ninguna.
- **Consistencia**: Después de una transacción la base de datos estará en un estado válido y consistente.
- **Aislamiento**: Cada transacción está aislada del resto de transacciones y que el acceso a los datos se hará de forma exclusiva. Si una transacción que quiere acceder de forma concurrente a los datos que están siendo utilizados por otra transacción, no podrá hacerlo hasta que la primera haya terminado.
- **Durabilidad**: Los cambios que realiza una transacción sobre la base de datos son permanentes.

Una sentencia simple de SQL es una transacción autocompletada (autocommit)

```
UPDATE Products SET UnitPrice=20 WHERE ProductName ='Chai'
```

Este tipo de sentencias se escriben en el fichero de transacciones de la base de datos y se realizan los cambios en la base de datos. En caso de problemas algunos motores de datos automaticamente retornan al estado anterior.

Cuando se deben generar varias sentencias como parte de un proceso es necesario ejercer control sobre estas acciones. Es decir, debe cumplirse el todo o nada.

1.6.1 Ejemplos

```
-- mysql


START TRANSACTION;

UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;

UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;

COMMIT;


-- postgresql

BEGIN TRANSACTION;

UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;

UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;

COMMIT;
```

```
-- estableciendo un punto de retorno en my_savepoint en caso de error
BEGIN;
UPDATE accounts SET balance = balance - 100.00
   WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
   WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
   WHERE name = 'Wally';
COMMIT;
```

1.8 Procedimientos

Los procedimientos[3], más comúnmente conocidos como **procedimientos almacenados (Store Procedure)** son un conjunto de instrucciones destinados a ejecutar un proceso específico. Estos pueden aceptar parámetros de entrada y salida, y pueden devolver ya sea un valor o un conjunto de registros. Los motores más utilizados de bases de datos como Oracle, MySQL, PostgreSQL y otros, permiten el uso de estos procedimientos.

En cada petición SQL se establece una comunicación entre el cliente que genera el SQL y el servidor que traduce esta petición para generar los resultados esperados y los transmite por la red hacia el cliente, cada petición entonces generará más tráfico. Una forma de reducir este tráfico y que varias interacciones se generen dentro del servidor antes de entregar el resultado final al cliente es mediante el uso de procedimientos almacenados (PA).

Como el PA se ejecuta en el servidor de base de datos, todas las instrucciones se ejecutan directamente en el servidor y una vez obtenidos los resultados estos son enviados al cliente. Además, si se realiza un cambio al PA este se activará para todos los clientes que lo utilicen, centralizando la lógica empresarial de la base de datos.

Estos PA entonces tienen la función primordial de hacer más eficiente el uso de la base de datos, ya que podemos generar una serie de procesos encadenados que se activen dentro del servidor, en lugar de realizar estos procesos uno a uno, donde cada proceso además significa una interrelación cliente y servidor lo que conlleva a un mayor tráfico de datos y tiempo.

Los PA se pueden escribir en varios lenguajes de programación, esto depende del motor utilizado

El lenguaje de procedimientos es similar al PL/SQL de Oracle y está disponible por defecto en cada instalación de PostgreSQL. Se utiliza para:

- Creación de procedimientos, funciones y disparadores
- Generar estructuras de control
- Realización de operaciones y cálculos complejos

Este lenguaje es estructurado en forma de bloques.

```
CREATE [ OR REPLACE ] FUNCTION

nombre_funcion([  [ argnombre ] argtipo ])

RETURNS tipo AS $$


[ DECLARE ]

 [ declaraciones de variables ]


BEGIN

  codigo

END;
```

```
CREATE OR REPLACE FUNCTION ejemplo(integer) RETURNS integer AS $$

BEGIN

 RETURN $1;

END;
```

1.8.1 ¿Para qué se usan los procedimientos almacenados?

En primer lugar, para mejorar el rendimiento de la aplicación. Cada ejecución se realiza en el servidor de la base de datos sin interacción con el cliente ni tener que procesar resultados intermedios.

Para simplificar la aplicación. Al delegar algunas funciones dentro de la base de datos y ahorrar por tanto complejidad en el desarrollo de la aplicación. Al tener la lógica de proceso dentro de la base de datos, esta se aplicará en forma independiente a la aplicación utilizada desde el lado del cliente. Por tanto nuestra capacidad de conocer a cabalidad nuestro motor de datos puede permitir alivianar muchas tareas complejas en la aplicación cliente.

NOTA: El lado negativo de esta propuesta reside en que, al vincular ciertos procesos a la base de datos utilizada, condiciona que la aplicación cliente debe mantenerse vinculada a esta y no podrá fácilmente ser utilizada con otro motor de datos.

Cuando tenemos relaciones entre tablas con la condición ON CASCADE, en que si eliminamos un registro en una tabla, posiblemente podemos eliminar a su vez los registros asociados en otra tabla y así sucesivamente. Pero con PA podemos establecer nuevas rutinas de relaciones: podremos establecer que al actualizar un registro de una tabla se realicen operaciones aritméticas en otros registros asociados, eliminar algunos, actualizar otros, y así sucesivamente. El cielo es el límite o nuestra capacidad de manejar procesos complejos, lo que ocurra primero.

1.8.2 Ejemplos

-- el tipico Hola mundo SQL Server

CREATE PROCEDURE HolamundoProcedure

AS

PRINT 'Hola mundo'

-- ejecutarlo...

exec HolamundoProcedure

1.9 Funciones

A diferencia de los procedimientos almacenados, las funciones siempre deben devolver un valor, y tiene solo parámetros de entrada y una cláusula de retorno que debe estar declarado en el encabezado.

Las funciones son seguras al estar dentro de la base de datos, de manera que oculta detalles a la aplicación.

-- formato de declaración de funcion. entre [] indica opcional.

CREATE [**OR REPLACE**] **FUNCTION** nombre_de_funcion(argumento **IN** tipo_de_dato)

RETURN tipo_de_dato

AS

BEGIN

codigo

END [nombre_de_funcion]

Nota

Un procedimiento almacenado (PA) puede llamar a funciones propias de SQL o de tipo UDF, pero una función no puede llamar a un PA.

1.9.1 Ejemplos

```
--cálculo de área de un círculo dado el radio
CREATE OR REPLACE FUNCTION area_circulo(radio IN NUMBER)
RETURN NUMBER
IS
   pi NUMBER := 3.14;
   area NUMBER;
BEGIN
  area := pi * POWER(radio, 2);
  RETURN area;
END area_circulo;


-- llamada
SELECT area_circulo(45)
-- regresa todas los articulos de la tabla itemes
CREATE OR REPLACE FUNCTION GetAllItemes()
RETURNS itemes
LANGUAGE SQL
AS
$$
  SELECT * FROM itemes;
$$


-- llamada
SELECT GetAllItemes()
-- el Hola mundo SQL server
```

```
CREATE FUNCTION dbo.HolamundoFunction()
RETURNS varchar(20)
AS
BEGIN
    RETURN 'Hello world'
END


-- ejecutarlo
SELECT HolamundoFunction() as resultado
```

-- Toma como argumento una fecha en formato date y la convierte en texto con un formato especifico de 'DD-MM-YYYY' (ej: 25-12-2022). Para ello usa la funcion propia de PostgreSQL 'to_char' para la conversion en texto.

-- el formato de fecha en las bases de datos se almacenan por defecto en el formato 'YYYY-MM-DD' (ej: 2022-12-25)

```
CREATE OR REPLACE FUNCTION date_format(fecha date)
RETURNS text
LANGUAGE sql
AS
$function$
  SELECT to_char($1, 'DD-MM-YYYY')
$function$;
```

-- Igual a la función anterior pero est vez acepta un formato de fecha que incluye la hora

```
CREATE OR REPLACE FUNCTION date_format(fecha timestamp without time zone, formato character varying)
 RETURNS text
 LANGUAGE sql
AS
```

```
$function$

SELECT to_char($1, 'DD-MM-YYYY')

$function$;


-- Esta vez se incluye un argumento adicional a la fecha que fuerza el formato de fecha de salida.


CREATE OR REPLACE FUNCTION date_format(fecha date, formato character varying)

 RETURNS text

 LANGUAGE sql

AS

$function$

  SELECT to_char($1, 'DD-MM-YYYY')

$function$;

-- Esta vez la función toma un texto con un formato de fecha y lo convierte en una fecha real. Utiliza la función estandar 'to_date' para la conversion de formato.


CREATE OR REPLACE FUNCTION public.date_format2(text)

RETURNS date

LANGUAGE sql

AS

$function$

SELECT to_date($1, 'DD-MM-YY')

$function$;


-- Esta vez toma una fecha en modo de texto y la transforma en un formato de fecha propio de postgresql, con formato largo.


CREATE OR REPLACE FUNCTION public.date_formatpg(text)

RETURNS date
```

```
LANGUAGE sql
AS
$function$
SELECT to_date($1, 'YYYY-MM-DD')
$function$;


-- Esta vez se controla si la fecha es nula, en cuyo caso devolverá ''.
CREATE OR REPLACE FUNCTION pgdate(text)
 RETURNS date
 LANGUAGE sql
AS
$function$
SELECT to_date(NULLIF($1,''), 'DD-MM-YYYY')
$function$;


-- idem anterior pero recibiendo una fecha.


CREATE OR REPLACE FUNCTION pgdate(date)
 RETURNS date
 LANGUAGE sql
AS $function$
SELECT to_date(NULLIF(to_char($1,'YYYY-MM-DD'),''),'YYYY-MM-DD')
$function$;
```

1.9.2 Procedimientos vs Funciones

- Un procedimiento no puede devolver un valor, una función sí (aunque ambos puedan devolver datos en parámetros `OUT` e `IN OUT`
- De lo anterior podemos deducir que las funciones se usan como parte de una expresión (`campo1 * función(campo2)`), los procedimientos no.
- La instrucción `return` devuelve el control al programa que la llama y entrega los resultados en ella, en el procedimiento devuelve el control al programa que la llamó, pero no devuelve un valor.

- Las funciones pueden ser llamadas desde instrucciones SQL (`SELECT`, `UPDATE`, `DELETE`, etc) y los procedimientos no (deben ser llamados de manera independiente mediante un CALL).
- Procedimientos: Ejecutar una serie de consultas, modificaciones en diferentes tablas o cálculos entre ellas que serían complejas en una única instrucción SQL y no requiere de un valor de retorno.
- Funciones: Cálculo de un valor que será usado en una consulta SQL (como campo de un resultado `SELECT`, asignación de valor en un `UPDATE` o como filtro en un `WHERE`) a partir de parámetros de entrada, ya sea accediendo a datos de diferentes tablas o bien a partir únicamente de los parámetros de entrada.

1.10 Disparadores (Triggers)

Los disparadores se encargan de activar una determinada function cuando ocurre un cierto evento ya sea en una tabla, vista o una tabla foránea. Estos disparadores se activan cuando se modifica la data mediante cualquier evento de una sentencia DML (Data Manipulation Language), esto incluye por cierto INSERT, UPDATE, DELETE o TRUNCATE. Se utilizan para ejecutar reglas de negocio en los datos, auditar o validar datos.

-- estructura de un trigger

CREATE TRIGGER triggername

{BEFORE | AFTER} {INSERT | UPDATE| DELETE }

ON tablename **FOR EACH ROW**

codigo;

--

CREATE [**OR REPLACE**] **TRIGGER** nombre_de_trigger

BEFORE -- contexto de tiempo

INSERT -- acción

ON nombre_de_tabla

FOR EACH ROW

BEGIN

codigo

END;

El contexto de tiempo aplica a una de las siguientes situaciones:

BEFORE: Solo se aplica a tablas y antes de verificar las restricciones. Es útil para chequear las restricciones de datos en aquellas situaciones donde no se puede verificar la integridad de ellos mediante las referencias.

AFTER: Solo aplica en tablas y después de realizar las operaciones. Se emplea para ejercer cambios en otras tablas en cascada.

INSTEAD OF: Se aplica en vistas, para hacerlas actualizables.

Eventos de trigger para tablas.

Evento	SQL	Nivel
BEFORE	DELETE	FOR EACH ROW
BEFORE	DELETE	
BEFORE	INSERT	FOR EACH ROW
BEFORE	INSERT	
BEFORE	UPDATE	FOR EACH ROW
BEFORE	UPDATE	
AFTER	DELETE	FOR EACH ROW
AFTER	DELETE	
AFTER	INSERT	FOR EACH ROW
AFTER	INSERT	
AFTER	UPDATE	FOR EACH ROW
AFTER	UPDATE	
INSTEAD OF	DELETE	FOR EACH ROW
INSTEAD OF	DELETE	
INSTEAD OF	INSERT	FOR EACH ROW

Evento	SQL	Nivel
INSTEAD OF	INSERT	
INSTEAD OF	UPDATE	FOR EACH ROW
INSTEAD OF	UPDATE	