Resumen SQL

Programacion Frontend

Julio 2023

Índice

1. Clausulas

1.1. Clausula SELECT

`SELECT * FROM users` se utiliza para seleccionar todo desde la tabla `users`. Ahora, vamos a insertar en la tabla users los campos (columnas) name, surname y age de la siguiente manera:

```
INSERT INTO users (name,surname,age) VALUES ('Ezequiel', 'Rueda Sanchez', '24')
```

Observamos que al ejecutar esta instrucción obtenemos:

`'Result: query executed successfully. Took 5ms'`.

Esto es así porque cualquier operación (en este caso `INSERT`) que realicemos con SQL la toma como una consulta.

Ahora si ejecutamos `SELECT * FROM users` nos devuelve la siguiente tabla:

Name	Surname	Age
Ezequiel	Rueda Sanchez	24

También puedo agregar varios usuarios al mismo tiempo de la siguiente manera:

```
INSERT into users (name, surname, age) VALUES ('Pedro', 'Rodriguez', 26), ('Carlos', 'Sanchez', 18),
('Francisco', 'Alvarez', 52)
```

Ahora, ejecuto `SELECT * FROM users` para ver el nuevo estado de la tabla.

Al ejecutar `SELECT age FROM users`, filtramos la tabla para que solo nos devuelva el campo `age` con sus respectivos registros.

Ahora insertamos el registro `('Francisco', 'Alvarez', 52)` y tenemos dos registros iguales.

¿Como los podemos diferenciar?

No hay manera de diferenciarlos. Para eso, utilizamos los **identificadores**. Tenemos los *primary keys* y los *foreign keys*.

Para ello, vamos a modificar la tabla colocando un nuevo campo llamado `id_user` de tipo integer.

Ahora, al mostrar la tabla (`SELECT * FROM users`), observamos que todos los registros tienen el campo `id_user` con `NULL`.

Entonces, vamos a borrar todos los registros ejecutando `DELETE FROM users`.

Ahora, vuelvo a modificar la tabla y puedo aplicar la opción `AI` (Auto Increment) para que cada registro sea unico y automáticamente se me aplica la opción `PK` (Primary Key) porque nos garantiza una integridad referencial, es decir, podemos hacer referencias a registros sin tener valores duplicados.

Por lo tanto, ahora a medida que vayamos insertando registros el id se va a auto incrementar en uno para que no haya ninguna posibilidad de que dos registros sean exactamente iguales. Luego, armamos una tabla nueva que llamamos `turnos_medicos` donde insertamos los campos `id_turno`, `profesional`, `id_user` (F(oreign) K(ey) o clave foranea), `motivo` y `horario`.

Decimos que una clave es foránea en una tabla cuando hace referencia a una clave primaria (PK) de otra tabla y se le debe colocar el mismo nombre tanto en la tabla como en la tabla donde es PK.

De ahora en adelante, vamos a trabajar con la base de datos de **Northwind**.

Supongamos que queremos cambiar el nombre del campo `LastName`. No lo puedo hacer desde la tabla pero puedo modificarlo localmente con `SELECT LastName AS surname FROM Employees`.

De ahora en adelante, esta renombrado localmente como `surname` aunque en la tabla el campo sigue siendo `LastName`.

También podemos renombrarlo varios campos mediante comas. Por ejemplo:

```
SELECT LastName AS surname, FirstName AS name FROM Employees
```

Por lo tanto, utilizamos la cláusula `AS` para que una tabla/campo sea mas declarativo o descriptivo.

Ahora, volvemos a ver toda la de products ejecutando `SELECT * FROM Products`.

Luego, quiero ordenar la tabla pero según el precio (campo `PRICE`) de forma creciente. Sabemos que por defecto, los productos están ordenados según el numero de `ID` porque son autoincrementables. Para ello ejecutamos las siguientes lineas:

```
SELECT * FROM Products ORDER BY price
```

También podríamos ejecutar `SELECT * FROM Products ORDER BY price ASC`. Y obtenemos el mismo resultado.

Si queremos ordenar los precios de forma decreciente, ejecutamos las siguientes lineas:

```
SELECT * FROM Products ORDER BY price DESC
```

`ASC` y `DESC` funcionan tanto para los precios como para el campo text `ProductName`. En este caso, por defecto o `ASC` ordena alfabeticamente de menor a mayor (de la A a la Z) y `DESC` ordena alfabeticamente de mayor a menor (de la Z a la A).

Notemos que la tabla ordena de acuerdo a la siguiente jerarquía:

```
NULL > NUMEROS > CARACTERES ESPECIALES > CARACTERES COMUNES
```

Por lo tanto, si agregamos un registro `NULL` en la tabla de `Products` y ejecutamos:

```
SELECT * FROM Products ORDER BY ProductName ASC
```

El registro `NULL` se convierte en el primero de la tabla.

Ahora, si quiero colocar todos los registros NULL al final de la tabla ejecuto:

```
SELECT * FROM Products ORDER BY ProductName ASC NULLS LAST
```

`ASC` esta ordenando de forma creciente, es decir, los `NULLs` van al principio pero al ejecutar `NULLs LAST` estoy indicando que los voy a colocar al final de la tabla.

Ahora, si quiero colocar todos los registros NULL al principio de la tabla cuando estoy ordenando de forma descendente ejecuto:

```
SELECT * FROM Products ORDER BY ProductName DESC NULLS FIRST
```

`ASC` esta ordenando de forma creciente, es decir, los `NULLs` van al principio pero al ejecutar `NULLs LAST` estoy indicando que los voy a colocar al final de la tabla.

Ahora, si queremos ordenar la tabla de una manera `RANDOM` ejecutamos:

```
SELECT * FROM Products ORDER BY RANDOM()
```

Ahora, si ejecutamos:

```
SELECT * FROM Products ORDER BY ProductName, SupplierID DESC
```

Se ordena primero respecto a `ProductName` y en caso de que los registros sean iguales, en base a `SupplierID`. Ahora, si queremos eliminar todos los repetidos por ejemplo de `ProductName` ejecutamos:

```
SELECT DISTINCT ProductName FROM Products
```

Con esto eliminamos todas las repeticiones de `NULL` y solo nos quedamos con una (con la primera aparición).

Ademas, podemos ordenar los registros del campo ProductName ejecutando lo siguiente:

```
SELECT DISTINCT ProductName FROM Products ORDER BY ProductName DESC
```

Por lo tanto, primero solo se deja la primera aparición de cada producto y luego se ordena los mismos de manera decreciente.

1.2. Clausula WHERE

Supongamos que queremos ver cual es el nombre asociado al `productId` numero 14. Entonces, comenzamos ejecutando:

`SELECT ProductName FROM Products` obteniendo el listado de productos.

Ahora si queremos ver cual es el nombre asociado al productId numero 14 ejecutamos:

```
SELECT ProductName FROM Products WHERE ProductID = 14
```

Si queremos obtener toda la información del registro con `ProductID = 14` ejecutamos:

```
SELECT * FROM Products WHERE ProductID = 14
```

Ahora, supongamos que solo tenemos un presupuesto de 40USD y quiero ver que productos puedo comprar. Para ello, ejecuto lo siguiente:

```
SELECT * FROM Products WHERE Price < 40
```

Ahora, supongamos que queremos eliminar el registro `NULL` con `ProductID = 80`. Para ello, ejecutamos lo siguiente:

```
DELETE FROM Products WHERE ProductID = 80
```

Observemos que no es necesario escribir `*` luego del `DELETE`.

1.3. Clausula UPDATE

Supongamos que queremos el precio del producto `Chais` (`ProductID = 1`) que es de 18USD por 20USD. Para ello, ejecutamos:

```
UPDATE Products SET Price = 20 WHERE ProductId = 1
```

Luego, ejecutamos `SELECT * FROM Products` para ver el nuevo estado de la tabla.

Ahora, si queremos modificar el valor de mas de un campo nos alcanza con enumerar el campo seguido de una coma (,) y listo.

Por ejemplo, `UPDATE Products SET Price = 20, ProductName = 'Chais modificado' WHERE ProductId = 1`

Para modificar el campo `Price` y `ProductName`.

1.4. Clausula AND

Ahora, vamos a la tabla de clientes (Customers). Entonces, si queremos consultar por aquellos que se encuentran entre el `CustomerId 50` inclusive y el `CustomerId 55` exclusive, utilizamos la clausula `AND` de la siguiente manera:

```
SELECT * FROM Customers WHERE CustomerId >= 50 AND CustomerId < 55
```

1.5. Clausula OR

Ahora, vamos a la tabla de empleados (Employees) aplicando `SELECT * FROM Employees`.

Luego, queremos ver los datos de las empleadas Nancy o Anne. Para ello, aplicamos `OR` que funciona como union, es decir:

```
SELECT * FROM Employees WHERE FirstName = "Nancy" OR FirstName = "Anne"
```

1.6. Clausula NOT

Volviendo a la tabla de productos (Products), queremos acceder a todos aquellos productos donde su precio no es mayor a 40. Para ello, utilizamos la cláusula `NOT` de la siguiente manera:

```
SELECT * FROM Products WHERE NOT Price > 40
```

1.7. Clausula LIMIT

Ahora, volvemos a la tabla Customers y queremos obtener los primeros 5 clientes a partir del CustomerID 50 y que el país del cliente no sea Alemania (Germany). Entonces, usamos la cláusula LIMIT para realizar esta solicitud de la siguiente manera:

```
SELECT * FROM Customers WHERE CustomerID >= 50 AND NOT Country = "Germany"LIMIT 5
```

Finalmente, podemos ordenar esta solicitud aplicando `ORDER BY` de la siguiente manera:

```
SELECT * FROM Customers WHERE CustomerID >= 50 AND NOT Country = "Germany.ºRDER BY RANDOM() LIMIT 5
```

1.8. Cláusula GROUP BY y HAVING

La clausula **GROUP BY** se utiliza para agrupar uno o varios registros según uno o varios valores de las columnas (campos).

Supongamos que queremos calcular el precio promedio de los productos de acuerdo a los proveedores y ordenarlos de manera decreciente. Para ello ejecutamos lo siguiente:

```
SELECT SupplierID, ROUND(AVG(Price)) as promedio from Products GROUP BY SupplierID ORDER BY promedio DESC
```

Ahora queremos seleccionar el nombre del producto (`ProductName`) también, ejecutamos lo siguiente:

```
SELECT ProductName, SupplierID, ROUND(AVG(Price)) as promedio from Products GROUP BY SupplierID ORDER BY promedio DESC
```

Observemos que el `ProductName` que retorna es aquel que figura primero con el `SupplierID` correspondiente.

La clausula `HAVING` nos sirve para realizar filtrado de grupos. A diferencia de la clase `WHERE` que nos sirve para realizar filtrado de registros.

Entonces, si queremos calcular el promedio de los productos de acuerdo a su `SupplierID`, agruparlos y mostrar aquellos en los que el promedio es mayor a 40 ejecutamos lo siguiente:

```
SELECT SupplierID, ROUND(AVG(Price)) as promedio from Products GROUP BY SupplierID HAVING promedio > 40
```

Observemos que aca no podemos usar la clausula `WHERE` porque queremos mostrar el promedio que es resultante de una función de agregación `ROUND(AVG(Price))` y no de un registro.

2. Operadores

2.1. Operador !=

Es un operador de comparación.

2.2. Operador BETWEEN

El operador **Between** es un operador de comparación que se utiliza para seleccionar valores en un rango especifico.

Supongamos ahora que queremos seleccionar todos aquellos productos donde su precio esta en el rango de 20 a 40 USD.

Para ello ejecutamos la siguiente linea:

```
SELECT * FROM Products WHERE Price BETWEEN 20 AND 40
```

También podemos combinarlo con los operadores `AND` y `OR` como por ejemplo:

```
SELECT * FROM Products WHERE Price BETWEEN 20 AND 40 AND CategoryID = 6
```

El operador BETWEEN también nos sirve para utilizarlo con fechas. Por ejemplo, si queremos filtrar a aquellos empleados que nacieron entre el año 1960 y 1970 ejecutamos lo siguiente:

```
SELECT * FROM Employees WHERE BirthDate BETWEEN "1960-0-1.ªND "1970-0-1"
```

Observamos también que los limites son inclusivos y que siempre se pasan del limite menor al limite mayor.

2.3. Operador LIKE

El operador **Like** es un operador de comparación utilizado para buscar y filtrar registros en función de ciertos patrones de cadena de texto.

Por ejemplo, si queremos filtrar todos los empleados con apellido `Fuller` podemos ejecutar lo siguiente:

```
SELECT * FROM Employees WHERE LastName LIKE "Fuller"
```

Esencialmente, utilizar `=` y `LIKE` es lo mismo pero `LIKE` tiene ciertos caracteres especiales que nos permiten realizar lo siguiente:

`SELECT * FROM Employees WHERE LastName LIKE "F%"`. En este caso, estamos realizando una búsqueda de aquellos apellidos que comienzan con `F` pero no me importa como siguen.

`SELECT * FROM Employees WHERE LastName LIKE "%r"`. En este caso, estamos realizando una búsqueda de aquellos apellidos que terminan con r pero no me importa como empiezan.

`SELECT * FROM Employees WHERE LastName LIKE "%a%"`. En este caso, estamos realizando una búsqueda de aquellos apellidos que tienen la letra a en alguna parte del `LastName`.

`SELECT * FROM Employees WHERE LastName LIKE #Full__"`. En este caso, estamos realizando una búsqueda de aquellos apellidos que comienzan con `Full` y tienen dos caracteres mas pero no me interesan cuales son. Esa es la función del caracteres especial.

2.4. Operador IS NULL y Operador IS NOT NULL

Supongamos ahora que queremos obtener unicamente aquellos productos que no son nulos y ordenarlos ascendentemente por `ProductName`. Para ello, ejecutamos lo siguiente:

```
SELECT * FROM Products WHERE ProductName IS NOT NULL ORDER BY ProductName ASC
```

Si queremos obtener unicamente aquellos productos que son nulos y ordenarlos descendentemente por ProductName, ejecutamos lo siguiente:

```
SELECT * FROM Products WHERE ProductName IS NOT NULL ORDER BY ProductName DESC
```

2.5. Operador IN

Introducción a las subconsultas.

Supongamos que queremos seleccionar los productos de los proveedores 3,4,5 y 6. Una opción es ejecutar:

```
SELECT * FROM Products WHERE SupplierID = 3 OR SupplierID = 4 OR SupplierID = 5 OR SupplierID = 6
```

La opción mas recomendada es usar el operador `IN` (funciona como un pertenece en conjuntos), es decir, podemos ejecutar:

```
SELECT * FROM Products WHERE SupplierID IN (3,4,5,6)
```

3. Funciones de agregación

Las **funciones de agregación** nos permiten agregar datos, resumirlos e incluso trabajar con estadísticas sobre los datos.

Supongamos que quiero contar la cantidad de nombres que hay en la tabla de empleados y por ende, la cantidad de empleados que tengo. Entonces, ejecuto:

```
SELECT count(firstName) FROM Employees
```

A esto lo permite la función `count()` que cuenta la cantidad de registros que tiene asociado un campo.

Por defecto, me devuelve el resultado con el nombre del campo `count(firstName)` pero si lo quiero renombrado utilizo `AS`, es decir:

```
SELECT count(firstName) AS Cantidad_de_nombres FROM Employees
```

Ahora, si quiero sumar todos los precios de la tabla Products utilizo la función SUM(), es decir:

```
SELECT SUM(Price) AS Suma_precios FROM Products
```

Si queremos calcular el promedio de los precios, utilizo la función `AVG()` de la siguiente manera:

```
SELECT AVG(Price) AS promedio FROM Products
```

Si queremos redondear a entero el promedio de los precios, utilizo la función `ROUND()` de la siguiente manera:

```
SELECT ROUND(AVG(Price)) AS promedio_redondeado FROM Products
```

Ahora, si quiero redondear por dos decimales ejecuto:

```
SELECT ROUND(AVG(Price), 2) AS promedio_redondeado_dos_unidades FROM Products
```

Ahora, si quiero calcular el producto de menor precio utilizo **MIN**:

```
SELECT MIN(Price) FROM Products
```

Ademas, si quiero mostar el nombre del producto de menor precio tambien ejecuto:

```
SELECT ProductName, MIN(Price) FROM Products
```

Ahora, si quiero calcular el producto de mayor precio utilizo **MAX**:

```
SELECT MAX(Price) FROM Products
```

4. Subconsultas

Ahora, el objetivo va a ser relacionar dos tablas. En este caso, vamos a querer relacionar la tabla de Productos (`Products`) con la tabla de detalles de la orden (`OrderDetails`) porque queremos obtener la cantidad de productos que se vendieron por cada producto y saber cual es el que genera mas ganancias. Con la tabla `OrderDetails` podemos calcular cual es producto (de acuerdo a su `ProductID`) que se vende mas veces pero eso no significa necesariamente que sea el que genera mas ganancia. Por ejemplo, puede haber un producto (supongamos `ProductID = 2`) que se haya vendido 20 veces y su precio sea de 1USD y puede haber otro producto (supongamos `ProductID = 5`) que se haya vendido solo 1 vez pero su precio sea de 200USD y observamos que es mayor la ganancia del producto con `ProductID = 5` que la del producto con `ProductID = 2`.

Entonces, para resolver esto vamos a usar las **Subconsultas** porque en la tabla `OrderDetails` tengo el `ProductID` con la cantidad `Quantity` de cada producto vendido. Y en la tabla `Products` tengo el precio (Price) y el `ProductID` de cada producto para poder realizar precio con cantidad de ventas de cada producto.

Ejemplo

Comenzamos consultando los campos `ProductID` y `Quantity` de la tabla `OrderDetails` ejecutando lo siguiente:

```
SELECT ProductID, Quantity FROM OrderDetails
```

Ahora, supongamos que quiero ver el nombre del producto pero este campo no lo tengo en la tabla `OrderDetails` sino en la tabla `Products`. Y lo puedo acceder a través de su `ProductID` que se encuentra en ambas tablas.

Entonces, para realizar esto ejecutamos lo siguiente:

```
SELECT ProductID, Quantity, (SELECT ProductName from Products WHERE OrderDetails.ProductID = ProductID)
as nombre_producto FROM OrderDetails
```

Por lo tanto, devolvemos de la tabla `OrderDetails` los campos `ProductID`, `Quantity` y el campo resultante de la subconsulta que renombrados como `nombre_producto`.

El campo `nombre_producto` es el resultante de la subconsulta:

```
SELECT ProductName from Products WHERE OrderDetails.ProductID = ProductID
```

En este caso, por cada `ProductID` de la tabla `OrderDetails`, busca en la tabla `Products` el `ProductID` que coincida con el `ProductID` de la tabla `OrderDetails` y retorna el `ProductName` que se encuentra en la tabla `Products`.

Ahora, queremos ver cual es el producto que mas recaudo. Para ello, ejecutamos lo siguiente:

```
SELECT ProductID, SUM(Quantity) as total_vendido,
(SELECT Price from Products WHERE OrderDetails.ProductID = ProductID) as Price,
(SELECT ProductName from Products WHERE OrderDetails.ProductID = ProductID) as Name,
ROUND(SUM(Quantity)) * (SELECT Price from Products WHERE OrderDetails.ProductID = ProductID) as Total_reca
FROM OrderDetails
GROUP BY ProductID
ORDER BY Total_recaudado DESC
```

Ahora, si queremos filtrar aquellos productos donde sus precios sean mayores a 40USD y que no se muestre el campo `Price`, ejecutamos lo siguiente:

```
SELECT ProductID, SUM(Quantity) as total_vendido,
(SELECT ProductName from Products WHERE OrderDetails.ProductID = ProductID) as Name,
ROUND(SUM(Quantity)) * (SELECT Price from Products WHERE OrderDetails.ProductID = ProductID) as Total_reca
FROM OrderDetails
WHERE (SELECT Price from Products WHERE OrderDetails.ProductID = ProductID) > 40
GROUP BY ProductID
ORDER BY Total_recaudado DESC
```

Ahora, si queremos realizar la misma consulta pero también queremos mostrar el campo `Price`, ejecutamos lo siguiente:

```
SELECT ProductID, SUM(Quantity) as total_vendido,
(SELECT Price from Products WHERE OrderDetails.ProductID = ProductID) as Price,
(SELECT ProductName from Products WHERE OrderDetails.ProductID = ProductID) as Name,
ROUND(SUM(Quantity)) * (SELECT Price from Products WHERE OrderDetails.ProductID = ProductID) as Total_reca
FROM OrderDetails
WHERE Price >40
GROUP BY ProductID
ORDER BY Total_recaudado DESC
```

Ahora, si quiero obtener solo el campo `Price` de esta subconsulta, ejecuto lo siguiente:

```
SELECT Price FROM (
SELECT ProductID, SUM(Quantity) as total_vendido,
(SELECT Price from Products WHERE OrderDetails.ProductID = ProductID) as Price,
(SELECT ProductName from Products WHERE OrderDetails.ProductID = ProductID) as Name,
ROUND(SUM(Quantity)) * (SELECT Price from Products WHERE OrderDetails.ProductID = ProductID) as Total_reca
FROM OrderDetails
WHERE Price > 40
GROUP BY ProductID
ORDER BY Total_recaudado DESC)
```

Ejercicio 2

Ahora, supongamos que queremos ver a aquellos empleados que lograron vender mas cantidad de unidades que el promedio. Entonces, comenzamos viendo el nombre (`FirstName`) y apellido (`LastName`) de los empleados (`Employees`) ejecutando:

```
SELECT FirstName, LastName FROM Employees
```

Ahora, vamos a solicitar la cantidad vendida de cada uno de los empleados con la siguiente subconsulta:

```
SELECT FirstName, LastName, (
SELECT SUM(od.Quantity) FROM [Orders] o, [OrderDetails] od
WHERE o.EmployeeID = e.EmployeeID AND o.orderID = od.orderID
) as Unidades_totales
FROM [Employees] e
```

Ahora, como queremos seleccionar a aquellos empleados tales que las `Unidades_totales` sean mayor al promedio podriamos intentar ejecutar `WHERE Unidades_totales > AVG(Unidades_totales)` pero esto no se puede. Entonces, creamos una subconsulta virtual que vuelva a pedir la cantidad de `Unidades_totales`, calculamos el promedio y comparamos con `Unidades_totales` de la siguiente manera:

```
SELECT FirstName, LastName, (
SELECT SUM(od.Quantity) FROM [Orders] o, [OrderDetails] od
WHERE o.EmployeeID = e.EmployeeID AND o.orderID = od.orderID
) as Unidades_totales
FROM [Employees] e
WHERE Unidades_totales >(SELECT AVG(Unidades_totales) FROM (SELECT
(SELECT SUM(od.Quantity) FROM [Orders] o, [OrderDetails] od
WHERE o.EmployeeID = e2.EmployeeID AND o.orderID = od.orderID
) as Unidades_totales
FROM [Employees] e2
GROUP BY e2.EmployeeID
)
)
```

5. Joins

Los **Joins** son una operación que utilizamos en las bases de datos para poder combinar la información de dos o mas tablas en una base de datos pero que esa información se devuelva en una sola tabla.

5.1. Cross Join

Si ejecutamos:

```
SELECT * FROM Employees e CROSS JOIN Orders o
```

Obtenemos una sola tabla con los campos unidos de `Employees` y `Orders` de 1960 registros. Como la lista de empleados esta compuesta por 10 registros y la tabla de `Orders` esta compuesta por 196 registros, al aplicar un **Cros Join**, le asigna a cada empleado cada uno de los 196 registros y los combina en una sola tabla.

Esto es analogo a ejecutar `SELECT * FROM Employees e, Orders o`.

5.2. Inner Join (Join)

Ahora, si queremos que se unan los datos del empleado que vendió cada producto (se encuentran en `Orders`) ejecutamos lo siguiente.

5.3. Creación de Tablas

Para crear una Tabla directamente a partir de lineas de código, ejecutamos las siguientes lineas:

```
CREATE TABLE "Rewards" (recompensa) (
"RewardID" INTEGER,
"EmployeeID" INTEGER,
"Reward" INTEGER,
"Month" TEXT.
PRIMARY KEY("RewardID" AUTOINCREMENT)
)
```

En este caso `RewardID` es la clave primaria, `EmployeeID` es el empleado que recibe la recompensa, `Reward` indica la recompensa en USD y `Month` indica el mes.

Luego, indicamos que `RewardID` va a ser la clave primaria autoincrementable ejecutando:

```
PRIMARY KEY("RewardID" AUTOINCREMENT)
```

Finalmente, obtenemos la tabla con todos los campos declarados anteriormente vacios porque todavía no les insertamos datos.

Ahora, vamos a insertar 6 datos. 5 datos de empleados diferentes donde en cada mes un empleado distinto va a recibir un premio y un mes donde ningún empleado recibe un premio. Es decir:

```
INSERT INTO Rewards (EmployeeID, Reward, Month) VALUES

(3, 200, "January"),
(2, 180, "February"),
(5, 250, "March"),
(1, 280, "April"),
(8, 160, "March"),
(null, null, "June")
```

Entonces, si queremos unir la información de los empleados que recibieron los premios ejecutamos lo siguiente:

```
SELECT * FROM Employees e
INNER JOIN Rewards r ON e.EmployeeID = r.EmployeeID
```

Pero si queremos obtener el nombre, la recompensa y el mes ejecutamos lo siguiente:

```
SELECT FirstName, Reward, Month FROM Employees e
INNER JOIN Rewards r ON e.EmployeeID = r.EmployeeID
```

Por lo tanto, un `INNER JOIN` representa la intersección entre conjuntos.

5.4. Left Join

Ahora, si quiero ejecutar un **Left Join** ejecuto lo siguiente:

```
SELECT FirstName, Reward, Month FROM Employees e
LEFT JOIN Rewards r ON e.EmployeeID = r.EmployeeID
```

Por lo tanto, un **Left Join** representa la diferencia entre conjuntos (en este caso `Employees - Rewards`). Es decir, muestra a todos los empleados de la tabla `Employees` y solo les une la recompensa y el mes a aquellos en los que coincide su `ID`. Los demás campos los deja en `NULL`.

Ahora, si queremos que la diferencia sea (en este caso `Rewards - Employees`) ejecutamos:

```
SELECT FirstName, Reward, Month FROM Rewards r
LEFT JOIN Employees e ON e.EmployeeID = r.EmployeeID
```

Observemos que en SQLite no podemos ejecutar un **Right Join**, es decir:

```
SELECT FirstName, Reward, Month FROM Employees e
RIGHT JOIN Rewards r ON e.EmployeeID = r.EmployeeID
```

Entonces, lo que hacemos para lograr esto es invertir el orden de las tablas como se menciono anteriormente pero debemos aclararlo en un comentario.

Ahora, si queremos unir en una sola tabla el resultado de dos consultas utilizamos `UNION`. Es decir, ejecutamos:

```
SELECT FirstName, Reward, Month FROM Employees e
LEFT JOIN Rewards r ON e.EmployeeID = r.EmployeeID

SELECT FirstName, Reward, Month FROM Rewards r
LEFT JOIN Employees e ON e.EmployeeID = r.EmployeeID
```

Por lo tanto, estamos simulando un `FULL JOIN` uniendo un `LEFT JOIN` con una simulación de `RIGHT JOIN`.

5.5. Union

Union es una cláusula que se utiliza para combinar dos o mas tablas.

. `UNION ALL` devuelve la suma de las dos consultas. Por ejemplo, si tengo una tabla con 4 filas y otra con 3 filas, el resultado es una tabla con 7 filas donde los campos son coincidentes.

. `UNION` también devuelve la suma de las dos consultas pero solo deja la primera aparición de todas las filas que están repetidas.

6. Cardinalidad

La **carnalidad** en el contexto de base de datos se utiliza para especificar cual es la relación entre dos entidades (generalmente tablas).

Comencemos viendo el ejemplo de la base de datos de NorthWind que venimos estudiando.

. La tabla `Customers` se une con la tabla `Orders` mediante el `CustomerID`. En este caso la relación es `1:n` porque 1 orden puede haber sido hecho únicamente por un cliente (`customer`) y `n` el cliente puede realizar muchas (`n`) ordenes.

. La tabla `Employees` se une con la tabla `Orders` mediante el `EmployeeID`. En este caso la relación es `1:n` porque 1 orden puede haber sido hecho únicamente por un vendedor/empleado (`employee`) y `n` porque el empleado puede realizar muchas (`n`) ordenes/ventas.

. La tabla `Shippers` se une con la tabla `Orders` mediante el `ShipperID`. En este caso la relación es `1:n` porque 1 empresa solo se puede encargar del envío de un producto y `n` porque una empresa se encargar de muchos (`n`) envíos.

. La tabla `Orders` se une con la tabla `OrderDetail` mediante el `OrderID`. En este caso la relación es `1:n` porque los detalles de una orden se corresponden únicamente con 1 orden y `n` porque 1 orden puede tener muchos (`n`) detalles.

. La tabla `OrderDetails` se une con la tabla `Products` mediante el `ProductID`. En este caso la relación es `n:1` porque un detalle hace referencia únicamente a 1 producto y `n` porque muchos (`n`) productos puede hacen referencia a diferentes detalles de orden.

. La tabla `Products` se une con la tabla `Suppliers` mediante el `SupplierID`. En este caso la relación es `n:1` porque un producto solo puede estar relacionado a un proveedor (supplier) y `n` porque un proveedor puede proveer de varios (`n`) productos.

. La tabla `Products` se une con la tabla `Categories` mediante el `CategoryID`. En este caso la relación es `n:1` porque 1 producto solo puede estar relacionado a una categoría (category) y `n` porque un categoría puede estar asociada a varios (`n`) productos.

7. Normalización

La normalización es un proceso dentro del diseño de la base de datos que sirve para eliminar anomalías en los datos, hacer que la base de datos sea mas eficiente y poder hacer consultas mas efectivas.

8. Indices

Un **indice** tiene el objetivo de mejorar el rendimiento de las consultas en una base de datos organizándolos correctamente.

. Indices primarios o únicos (primary key por ejemplo): Los indices primarios no pueden tener valores nulos, es decir, es el único tipo de indice que no permite tener valores nulos

Por ejemplo, vamos a crear un indice en campo LastName de la tabla Employees de la siguiente manera:

```
CREATE INDEX name on Employees (FirstName)
```

El problema de esto es que el campo `FirstName` se convierte en un indice y puede admitir campos nulos y campos duplicados.

Si agregamos `UNIQUE` adelante de `INDEX`, crea un indice único para el nombre (`FirstName`). Es decir, cuando se quiera insertar un valor que se repita, la consulta no nos va a insertar el registro porque es un valor repetido o duplicado gracias al `UNIQUE`.

```
CREATE UNIQUE INDEX name on Employees (FirstName, LastName)
```

También puedo crear campos únicos combinados, es decir, los campos `FirstName` y `LastName` son únicos, es decir, cuando quiera insertar un campo donde el nombre y apellido son repetidos, me va a arrojar un error.

Ejemplo

Queremos consultar todos los productos que tuvieron mas de 10 ventas y que se vendieron luego de la fecha "`1996-07-04`". Es decir:

```
SELECT * FROM OrderDetails od JOIN Orders o WHERE o.OrderID = od.OrderID AND OrderDate > "1996-07-04"
AND od.Quantity > 10
```

Como utilizamos muchos los campos `OrderDate` y `Quantity`, creamos los siguientes indices:

```
CREATE INDEX idx_orderdetails_quantity ON OrderDetails (Quantity);
CREATE INDEX idx_orders_quantity ON Orders (OrderDate)
```

Luego, volvemos a ejecutar la consulta:

```
SELECT * FROM OrderDetails od JOIN Orders o WHERE o.OrderID = od.OrderID AND OrderDate > "1996-07-04"
AND od.Quantity > 10
```

Y observamos que no se produce una mejora en el rendimiento (sigue demorando entre 17ms y 20ms).

Ahora, si quiero eliminar los indices, ejecuto:

```
DROP INDEX idx_orderdetails_quantity;
DROP INDEX idx_orders_quantity
```

Ahora, si ejecutamos nuevamente la consulta:

```
SELECT * FROM OrderDetails od JOIN Orders o WHERE o.OrderID = od.OrderID AND OrderDate > "1996-07-04"
AND od.Quantity > 10
```

Vemos que demora entre 28ms y 31ms. Demora mas que antes de borrar los indices.

9. Vistas

Las vistas son tablas virtuales. Las vistas no almacenan datos sino que referencian a una consulta que nos devuelve una vista.

Ejemplo

Comenzamos consultando por aquellos productos donde el `ProductID` > `20` y lo ordenamos por el id del producto de forma descendente. Es decir, seleccionamos y ordenamos los productos del ultimo al primero que fue cargado. O sea:

```
SELECT * FROM Products WHERE ProductID >20 ORDER BY ProductID DESC
```

Ahora, si queremos obtener únicamente el id, name y el price ejecutamos:

```
SELECT ProductID, ProductName, Price FROM Products WHERE ProductID > 20 ORDER BY ProductID DESC
```

Por lo tanto, a esta tabla resultante le puede crear una **vista** de la siguiente manera:

```
CREATE VIEW Productos_simplificados AS SELECT ProductID, ProductName, Price FROM Products WHERE ProductID
> 20 ORDER BY ProductID DESC
```

Ahora, puedo ver esta tabla ejecutando simplemente `SELECT * FROM Productos_simplificados`

Por ultimo, si queremos eliminar una vista ejecutamos:

```
DROP VIEW IF EXISTS Productos_simplificados
```

La clausula `IF EXISTS` la ejecutamos para evitar tener errores porque por ejemplo si borramos la vista ejecutando:

```
DROP VIEW Productos_simplificados
```

Borramos la vista correctamente pero si volvemos a ejecutar `DROP VIEW Productos_simplificados`, arroja un error porque la vista ya fue eliminada. Entonces, con `IF EXISTS` nos aseguramos chequear si existe la vista antes de borrarla.

10. Funciones definidas por el usuario

Una función definida por el usuario es una función que puede ser utilizada por SQLite y esta creada para poder recibir la información que puede ser ejecutada en una consulta, es decir, la función toma esos valores, los procesa y devuelve una salida. Ademas, debe estar registrada en SQLite. A su vez, esta función no va a estar creada en SQLite sino en un lenguaje anfitrión como por ejemplo Python