Introducción a SQL

(http://sql.1keydata.com/es)

SQL (Structured Query Language) es un lenguaje de programación diseñado para almacenar, manipular y recuperar datos almacenados en bases de datos relacionales. La primera encarnación de SQL apareció en 1974, cuando un grupo de IBM desarrolló el primer prototipo de una base de datos relacional. Relational Software (luego se convirtió en Oracle) lanzó la primera base de datos relacional comercial.

Existen estándares para SQL. Sin embargo, el SQL que puede utilizarse en cada uno de las principales RDBMS actuales viene en distintas formas. Esto se debe a dos razones: 1) el estándar SQL es bastante complejo, y no es práctico implementar el estándar completo, y 2) cada proveedor de base de datos necesita una forma de diferenciar su producto de otros. En esta guía de referencia, dichas diferencias se señalarán cuando sea apropiado.

Este sitio de la guía de referencia SQL enumera los comandos SQL normalmente utilizados, y se divide en las siguientes secciones :

- **Comandos SQL**: Las instrucciones SQL básicas para almacenamiento, recuperación y manipulación de datos en una base de datos relacional.
- **Manipulación de Tabla**: Cómo se utilizan las instrucciones SQL para administrar las tablas dentro de una base de datos.
- **SQL Avanzado**: Comandos SQL avanzados.
- **Sintaxis SQL**: Una página única que enumera la sintaxis para todos los comandos SQL en esta guía de referencia.

Para cada comando, primero se presentará y explicará la sintaxis SQL, seguida por un ejemplo. Al final de esta guía de referencia, deberá tener una idea general de la sintaxis SQL. Además, deberá poder realizar consultas SQL utilizando la sintaxis apropiada. Según mi experiencia creo que el comprender lo básico de SQL es mucho más fácil que dominar todas las dificultades de este lenguaje de base de datos, y espero que también llegue a la misma conclusión.

Si está interesado en cómo recuperar datos utilizando SQL, le recomendamos que empiece con la sección **Comandos SQL**. Si está interesado en comprender cómo puede utilizarse SQL para manipular una tabla de base de datos, le recomendamos que comience con la sección **Manipulación de Tabla**. Si está buscando ayuda sobre un comando SQL específico, puede utilizar el **Mapa del Sitio** para encontrar el comando que está buscando.

I- SELECT (Introducción)

1. SELECT – FROM

- ¿Para qué utilizamos los comandos SQL? El uso común es la selección de datos desde tablas ubicadas en una base de datos. Inmediatamente, vemos dos palabras claves: necesitamos **SELECT** la información **FROM** una tabla. (Note que la tabla es un contenedor que reside en la base de datos donde se almacena la información. Para obtener más información acerca de cómo manipular tablas, consulte la **Sección Manipulación de Tabla**).

- Por lo tanto tenemos la estructura SQL más básica:

SELECT "nombre_columna" FROM "nombre_tabla"

- Para ilustrar el ejemplo anterior, suponga que tenemos la siguiente tabla:

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Podemos utilizar esta tabla como ejemplo a lo largo de la guía de referencia (esta tabla aparecerá en todas las secciones). Para seleccionar todos los negocios en esta tabla, ingresamos,

SELECT store_name **FROM** Store_Information

Resultado:

store_name
Los Angeles
San Diego
Los Angeles
Boston

Pueden seleccionarse los nombres de columnas múltiples, así como también los nombres de tablas múltiples.

2. DISTINCT

- La palabra clave **SELECT** nos permite tomar toda la información de una columna (o columnas) en una tabla. Esto, obviamente, significa necesariamente que habrá redundancias. ¿Qué sucedería si sólo deseamos seleccionar cada elemento **DISTINCT**? Esto es fácil de realizar en SQL. Todo lo que necesitamos hacer es agregar **DISTINCT** luego de **SELECT**.

- La sintaxis es la siguiente:

```
SELECT DISTINCT "nombre_columna"
FROM "nombre_tabla"
```

- Por ejemplo, para seleccionar todos los negocios distintos en la Tabla ***Store_Information***,

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Ingresamos,

```
SELECT DISTINCT store_name FROM Store_Information
```

Resultado:

store_name
Los Angeles
San Diego
Boston

3. WHERE

- Luego, podríamos desear seleccionar condicionalmente los datos de una tabla. Por ejemplo, podríamos desear sólo recuperar los negocios con ventas mayores a $1.000 dólares estadounidenses.

- Para ello, utilizamos la palabra clave **WHERE**. La sintaxis es la siguiente:

```
SELECT "nombre_columna"
FROM "nombre_tabla"
WHERE "condición"
```

- Por ejemplo, para seleccionar todos los negocios con ventas mayores a 1.000€ dólares estadounidenses en la Tabla ***Store_Information***,

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Ingresamos,

```
SELECT store_name
FROM Store_Information
WHERE Sales > 1000
```

Resultado:

store_name
Los Angeles

4. AND-OR

- En la sección anterior, hemos visto que la palabra clave **WHERE** también puede utilizarse para seleccionar datos condicionalmente desde una tabla. Esta condición puede ser una condición simple (como la que se presenta en la sección anterior), o puede ser una condición compuesta. Las condiciones compuestas están formadas por múltiples condiciones simples conectadas por **AND** u **OR**. No hay límites en el número de condiciones simples que pueden presentarse en una sola instrucción SQL.
- La sintaxis de una condición compuesta es la siguiente:

```
SELECT "nombre_columna"
FROM "nombre_tabla"
WHERE "condición simple"
      { [AND|OR]
       "condición simple" }+
```

{}+ significa que la expresión dentro de las llaves ocurrirá una o más veces.

[|] Indica que **AND** u **OR** pueden utilizarse indistintamente.

Además, podemos utilizar el símbolo paréntesis () para indicar el orden de la condición.

- Por ejemplo, podemos desear seleccionar todos los negocios con ventas mayores a 1000€ dólares estadounidenses o todos los negocios con ventas menores a 500€ dólares estadounidenses pero mayores a 275€ dólares estadounidenses en la Tabla ***Store_Information***,

Tabla ***Store_Information***

store_name	*Sales*	*Date*
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
San Francisco	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Ingresamos,

```
SELECT store_name
FROM Store_Information
WHERE Sales > 1000
OR (Sales < 500 AND Sales > 275)
```

Resultado:

store_name
Los Angeles
San Francisco

5. IN

- En SQL, hay dos usos de la palabra clave **IN**, y esta sección introduce aquél relacionado con la cláusula **WHERE**. Cuando se lo utiliza en este contexto, sabemos exactamente el valor de los valores regresados que deseamos ver para al menos una de las columnas.

- La sintaxis para el uso de la palabra clave **IN** es la siguiente:

```
SELECT "nombre_columna"
FROM "nombre_tabla"
WHERE "nombre_columna" IN ('valor1', 'valor2', ...)
```

El número de valores en los paréntesis pueden ser uno o más, con cada valor separado por comas. Los valores pueden ser números o caracteres. Si hay sólo un valor dentro del paréntesis, este comando es equivalente a

```
WHERE "nombre_columna" = 'valor1'
```

- Por ejemplo, podríamos desear seleccionar todos los registros para los negocios de Los Ángeles y San Diego en la Tabla ***Store_Information***,

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
San Francisco	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Ingresamos,

```
SELECT *
FROM Store_Information
WHERE store_name IN ('Los Angeles', 'San Diego')
```

Resultado:

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999

5. BETWEEN

- Mientras que la palabra clave **IN** ayuda a las personas a limitar el criterio de selección para uno o más valores discretos, la palabra clave **BETWEEN** permite la selección de un rango.

- La sintaxis para la cláusula **BETWEEN** es la siguiente:

```
SELECT "nombre_columna"
FROM "nombre_tabla"
WHERE "nombre_columna" BETWEEN 'valor1' AND 'valor2'
```

  Esto seleccionará todas las filas cuya columna tenga un valor entre 'valor1' y 'valor2'.

- Por ejemplo, podríamos desear seleccionar la visualización de toda la información de ventas entre el 06 de enero de 1999, y el 10 de enero de 1999, en la Tabla ***Store_Information***,

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
San Francisco	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Ingresamos,

```
SELECT *
FROM Store_Information
WHERE Date BETWEEN '06-Jan-1999' AND '10-Jan-1999'
```

Tenga en cuenta que la fecha puede almacenarse en diferentes formatos según las diferentes bases de datos. Esta guía de referencia simplemente elige uno de los formatos.

Resultado:

store_name	Sales	Date
San Diego	250 €	07-Jan-1999
San Francisco	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

6. LIKE

- **LIKE** es otra palabra clave que se utiliza en la cláusula **WHERE**. Básicamente, **LIKE** le permite hacer una búsqueda basada en un patrón en vez de especificar exactamente lo que se desea (como en **IN**) o determinar un rango (como en **BETWEEN**).
- La sintaxis es la siguiente:

```
SELECT "nombre_columna"
FROM "nombre_tabla"
WHERE "nombre_columna" LIKE {patrón}
```

{patrón} generalmente consiste en comodines. Aquí hay algunos ejemplos:

- 'A_Z': Toda línea que comience con 'A', otro carácter y termine con 'Z'. Por ejemplo, 'ABZ' y 'A2Z' deberían satisfacer la condición, mientras 'AKKZ' no debería (debido a que hay dos caracteres entre A y Z en vez de uno).
- 'ABC%': Todas las líneas que comienzan con 'ABC'. Por ejemplo, 'ABCD' y 'ABCABC' ambas deberían satisfacer la condición.
- '%XYZ': Todas las líneas que terminan con 'XYZ'. Por ejemplo, 'WXYZ' y 'ZZXYZ' ambas deberían satisfacer la condición.
- '%AN%': Todas las líneas que contienen el patrón 'AN' en cualquier lado. Por ejemplo, 'LOS ANGELES' y 'SAN FRANCISCO' ambos deberían satisfacer la condición.

- Digamos que tenemos la siguiente tabla:

Tabla ***Store_Information***

store_name	Sales	Date
LOS ANGELES	1500 €	05-Jan-1999
SAN DIEGO	250 €	07-Jan-1999
SAN FRANCISCO	300 €	08-Jan-1999
BOSTON	700 €	08-Jan-1999

Deseamos encontrar todos los negocios cuyos nombres contengan ‘AN’. Para hacerlo, ingresamos,

```
SELECT *
FROM Store_Information
WHERE store_name LIKE '%AN%'
```

Resultado:

store_name	Sales	Date
LOS ANGELES	1500 €	05-Jan-1999
SAN DIEGO	250 €	07-Jan-1999
SAN FRANCISCO	300 €	08-Jan-1999

7. ORDER BY

- Hasta ahora, hemos visto cómo obtener datos de una tabla utilizando los comandos **SELECT** y **WHERE**. Con frecuencia, sin embargo, necesitamos enumerar el resultado en un orden particular. Esto podría ser en orden ascendente, en orden descendente, o podría basarse en valores numéricos o de texto. En tales casos, podemos utilizar la palabra clave **ORDER BY** para alcanzar nuestra meta.

- La sintaxis para una instrucción **ORDER BY** es la siguiente:

```
SELECT "nombre_columna"
FROM "nombre_tabla"
[WHERE "condición"]
ORDER BY "nombre_columna" [ASC, DESC]
```

  - [] significa que la instrucción **WHERE** es opcional. Sin embargo, si existe una cláusula **WHERE**, viene antes de la cláusula **ORDER BY**
  - **ORDER BY ASC** significa que los resultados se mostrarán en orden ascendente, y **DESC** significa que los resultados se mostrarán en orden descendente. Si no se especifica ninguno, la configuración predeterminada es **ASC**.
  - Es posible ordenar por más de una columna. En este caso, la cláusula **ORDER BY**anterior se convierte en

```
ORDER BY "nombre1_columna" [ASC, DESC], "nombre2_columna"
[ASC, DESC]
```

Suponiendo que elegimos un orden ascendente para ambas columnas, el resultado se clasificará en orden ascendente según la columna 1. Si hay una relación para el valor de la columna 1, se clasificará en orden ascendente según la columna 2.

- Por ejemplo, podríamos desear enumerar los contenidos de la Tabla ***Store_Information*** según la suma en dólares, en orden descendente:

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
San Francisco	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Ingresamos,

```
SELECT store_name, Sales, Date
FROM Store_Information
ORDER BY Sales DESC
```

Resultado:

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
Boston	700 €	08-Jan-1999
San Francisco	300 €	08-Jan-1999
San Diego	250 €	07-Jan-1999

Además del nombre de la columna, podríamos utilizar la posición de la columna (según la consulta SQL) para indicar en qué columna deseamos aplicar la cláusula **ORDER BY**. La primera columna es 1, y la segunda columna es 2, y así sucesivamente. En el ejemplo anterior, alcanzaremos los mismos resultados con el siguiente comando:

```
SELECT store_name, Sales, Date
FROM Store_Information
ORDER BY 2 DESC
```

- Ejercicios:

1- Obtener un listado de las ventas según su cuantía, de mayor a menor valor y la ciudad asociada a cada venta

2- Obtener un listado ordenado de las ciudades y sus ventas, ordenadas por fecha

3- Obtener la cuantía de las ventas de las ciudades cuyo nombre empiece por 'S', ordenado por la fecha de venta (no mostrar la fecha de venta)

8. FUNCIONES DE GRUPO

- Ya que hemos comenzado trabajando con números, la siguiente pregunta natural a realizarse es si es posible hacer cálculos matemáticos con aquellos números, tales como sumas, o sacar un promedio. ¡La respuesta es sí! SQL tiene varias funciones aritméticas, y estas son:

  - **AVG** : media aritmética
  - **COUNT** : contar filas
  - **MAX** : máximo
  - **MIN** : mínimo
  - **SUM** : suma

  Nota: En SQL se pueden realizar operaciones aritméticas simples como suma (+) y resta (-) sobre datos individuales. También hay funciones de manejo de cadenas (concatenación, etc) sobre datos individuales. No confundir estos operadores y funciones con las funciones de grupo, que siempre se aplican a TODOS los datos de un grupo.

  Cada una de estas funciones recibe como argumento una columna, de manera que realiza la operación sobre la proyección de TODAS las filas de esa columna (en realidad se pueden aplicar a grupos de filas: ver GROUP BY)

- La sintaxis para el uso de funciones es,

```
SELECT "tipo de función"("nombre_columna")
FROM "nombre_tabla"
```

- Por ejemplo, si deseamos obtener el sumatorio de todas las ventas de la siguiente tabla,

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Ingresaríamos,

```
SELECT SUM(Sales) FROM Store_Information
```

Resultado:

SUM(Sales)
2750 €

2 750 € dólares estadounidenses representa la suma de todas las entradas de Ventas: 1500 € + 250 € + 300 € + 700 €.

- Ejercicios:

  1- Ciudad(es) y valor de venta de mayor cuantía

  2- Valor de la venta de menor cuantía, sin repetir

  3- Media de las ventas entre el 7 y el 10 de Enero de 1999 de las ciudades de San Diego y Los Angeles

9. COUNT

- Una función de grupo muy utilizada es **COUNT**. Esto nos permite **COUNT** el número de filas en una tabla determinada.
- La sintaxis es,

```
SELECT COUNT("nombre_columna")
FROM "nombre_columna"
```

También podría escribirse **COUNT(*)**, si resulta indiferente por que columna contar.

- Por ejemplo, si deseamos encontrar el número de entradas de negocios en nuestra tabla,

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700,00 €	08-Jan-1999

ingresamos,

```
SELECT COUNT(store_name)
FROM Store_Information
```

Resultado:

Count(store_name)
4

COUNT y **DISTINCT** pueden utilizarse juntos en una instrucción para determinar el número de las distintas entradas en una tabla. Por ejemplo, si deseamos saber el número de los distintos negocios, ingresaríamos,

```
SELECT COUNT(DISTINCT store_name)
FROM Store_Information
```

Resultado:

Count(DISTINCT store_name)
3

- Ejercicios:

1- Mostrar cuantas ventas se realizaron en 1999

2- Mostrar cuantas cantidades distintas existen en las ventas realizadas

10. GROUP BY

- Ahora regresamos a las funciones de grupo. ¿Recuerda que utilizamos la palabra clave **SUM** para calcular las ventas totales para todas las ciudades? ¿Y si quisiéramos calcular el total de ventas para *cada* ciudad? En este caso necesitamos hacer dos cosas: Primero, necesitamos asegurarnos de que hayamos agrupado juntas todas las filas de una misma ciudad (**GROUP BY** "store_name"); Segundo, debemos sumar, dentro de cada grupo, el valor de todas las ventas (columna "Sales" de todas las filas asociadas a esa ciudad),

- La sintaxis SQL correspondiente es,

```
SELECT "nombre1_columna", SUM("nombre2_columna")
FROM "nombre_tabla"
GROUP BY "nombre1-columna"
```

Nota: Observe que cuando se emplea la clausula **GROUP BY**, el nombre de la columna por la que se agrupa, debe aparecer tanto en la cláusula **GROUP BY** como en la cláusula **SELECT**. Además, si se va a invocar a alguna función de grupo para que realice alguna operación aritmética, se aplicará, normalmente, a otra columna.

- Ilustremos utilizando la siguiente tabla,

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Deseamos saber las ventas totales para cada negocio. Para hacerlo, ingresaríamos,

```
SELECT store_name, SUM(Sales)
FROM Store_Information
GROUP BY store_name
```

Resultado:

store_name	SUM(Sales)
Los Angeles	1800 €
San Diego	250 €
Boston	700 €

La palabra clave **GROUP BY** se utiliza cuando estamos seleccionado columnas múltiples desde una tabla (o tablas) y aparece al menos un operador aritmético en la instrucción **SELECT**. Cuando esto sucede, necesitamos **GROUP BY** todas las otras columnas seleccionadas, *es decir*, todas las columnas excepto aquella(s) que se operan por un operador aritmético.

- Ejercicios:

1- Obtener la media de las cuantías de las ventas realizadas cada día

2- Obtener las diferentes cuantías de ventas y para cada cantidad, las ciudades (sin repetir) que tienen alguna venta con esa cuantía

11. HAVING

- Otra cosa que la gente puede querer hacer es limitar el resultado según la suma correspondiente (o cualquier otra función de grupo). Por ejemplo, podríamos desear ver sólo los negocios con ventas totales mayores a 1500 €, dólares. En vez de utilizar la cláusula **WHERE** en la instrucción SQL, necesitamos utilizar la cláusula **HAVING**, que se reserva para imponer condiciones sobre el valor de funciones de grupo. La cláusula **HAVING** se coloca generalmente cerca del fin de la instrucción SQL. La instrucción SQL con la cláusula **HAVING** puede o no incluir la cláusula **GROUP BY**

  En cierto sentido, la cláusula **HAVING** aplicada sobre grupos y funciones de grupo, desempeña el mismo papel que la cláusula **WHERE** aplicada sobre todas las filas de la tabla: “filtrar” las filas para quedarnos solamente con aquellas que cumplan la condición indicada.

- La sintaxis es,

```
SELECT "nombre1_columna", SUM("nombre2_columna")
FROM "nombre_tabla"
GROUP BY "nombre1_columna"
HAVING (condición de función aritmética)
```

  Nota: La cláusula **GROUP BY** es opcional.

- En nuestro ejemplo, tabla ***Store_Information***,

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

ingresaríamos,

```
SELECT store_name, SUM(sales)
FROM Store_Information
GROUP BY store_name
HAVING SUM(sales) > 1500
```

Resultado:

store_name	SUM(Sales)
Los Angeles	1800 €

- Ejercicios: Obtener la media de las cuantías de las ventas realizadas cada día a partir del 7 de Enero de 1999, siempre y cuando esa media supero los 700 €

12. ALIAS

- Nos concentraremos ahora en el uso de alias. Hay dos tipos de alias que se utilizan con mayor frecuencia. Alias de columna y alias de tabla:

  - Resumiendo, los **alias de columna** existen para ayudar en la organización del resultado. En el ejemplo anterior, cualquiera sea el momento en que vemos las ventas totales, se enumeran como **SUM**(sales). Mientras esto es comprensible, podemos ver casos donde el título de la columna pueden complicarse (especialmente si incluye varias operaciones aritméticas). El uso de un alias de columna haría el resultado mucho más legible.

  - El segundo tipo de alias es el **alias de tabla**. Esto se alcanza al colocar un alias directamente luego del nombre de tabla en la cláusula **FROM**. Esto es conveniente cuando desea obtener información de dos tablas separadas (el término técnico es 'realizar uniones'). La ventaja de utiliza un alias de tablas cuando realizamos uniones es rápidamente aparente cuando hablamos de uniones.

- Antes de comenzar con las uniones, miremos la sintaxis tanto para el alias de columna como de tabla:

```
SELECT "alias_tabla"."nombre1_columna" "alias_columna"
FROM "nombre_tabla" "alias_tabla"
```

Brevemente, ambos tipos de alias se colocan directamente después del elemento por el cual generan el alias, separados por un espacio en blanco.

- Nuevamente utilizamos nuestra tabla, ***Store_Information***,

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Utilizamos el mismo ejemplo que en la sección **SQL GROUP BY**, salvo que hemos colocado tanto el alias de columna como el alias de tabla:

```
SELECT A1.store_name Store, SUM(A1.Sales) "Total Sales"
FROM Store_Information A1
GROUP BY A1.store_name
```

Resultado:

Store	Total Sales
Los Angeles	1800 €
San Diego	250 €
Boston	700 €

Note la diferencia en el resultado: los títulos de las columnas ahora son diferentes. Ese es el resultado de utilizar el alias de columna. Note que en vez de “Sum(sales)” de algún modo enigmático, ahora tenemos “Total Sales”, que es más comprensible, como título de columna. La ventaja de utilizar un alias de tablas no es fácil de ver en este ejemplo. Sin embargo, se tornará evidente en la siguiente sección

13. JOIN

- Digamos que tenemos las siguientes dos tablas:

Tabla ***Store_Information***

store_name	***Sales***	***Date***
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Tabla ***Geography***

region_name	**store_name**
East	Boston
East	New York
West	Los Angeles
West	San Diego

y queremos saber las ventas por región. Vemos que la tabla ***Geography*** incluye información sobre regiones y negocios, y la tabla ***Store_Information*** contiene información de ventas para cada negocio.

- Un primer intento podría consistir en realizar la siguiente consulta:

```
SELECT A1.region_name A1.store_name A2.store_name A2.Sales
FROM Geography A1, Store_Information A2
GROUP BY A1.region_name
```

  - Las primeras dos líneas le indican a SQL que seleccione cuatro campos; los dos primeros son los campo ***region_name*** y ***store_name*** de la tabla ***Geography*** (denominado REGIÓN), y el resto son los campos ***store_name*** y ***Sales*** de la tabla ***Store_Information***. Note como se utilizan los alias de tabla aquí: Geografía se denomina A1, e Información_Negocio se denomina A2.

- Esta consulta da como resultado la **unión cartesiana** de las tablas *Store_Information* y *Geography* (proyectando sobre las columnas *region_name* y *sales*). Las uniones cartesianas darán por resultado que de la consulta se arroje toda combinación posible de las dos tablas (o cualquiera que sea el número de tablas en la instrucción **FROM**). En este caso, una unión cartesiana resultaría en un total de 4x4 = Se presenta un resultado de16 filas.

Producto cartesiano ***Geography x Store_Information***

A1.region_name	***A1.store_name***	***A2.store_name***	***A2.Sales***
East	Boston	Los Angeles	1500 €
East	Boston	San Diego	250 €
East	Boston	Los Angeles	300 €
East	Boston	Boston	700 €
East	New York	Los Angeles	1500 €
East	New York	San Diego	250 €
East	New York	Los Angeles	300 €
East	New York	Boston	700 €
West	Los Angeles	Los Angeles	1500 €
West	Los Angeles	San Diego	250 €
West	Los Angeles	Los Angeles	300 €
West	Los Angeles	Boston	700 €
West	San Diego	Los Angeles	1500 €
West	San Diego	San Diego	250 €
West	San Diego	Los Angeles	300 €
West	San Diego	Boston	700 €

Como puede observarse, muchas de las combinaciones “no tienen sentido”; el producto cartesiano simplemente genera todas las posibles combinaciones de filas de las dos tablas.

- Las combinaciones que tienen sentido al combinar las dos tablas son aquellas en las que coinciden los valores de **A1.store_name** y **A2.store_name**. Es decir, hay que *reunir* las dos tablas combinando las filas en las que A1.store_name=A2.store_name. Eso se denomina ***join*** o ***reunión*** de dos tablas.

```
SELECT A1.region_name A1.store_name A2.store_name A2.Sales
FROM Geography A1, Store_Information A2

WHERE A1.store_name=A2.store_name
GROUP BY A1.region_name
```

El resulta de la consulta es el siguiente:

Join de ***Geography y Store_Information***

A1.region_name	*A1.store_name*	*A2.store_name*	*A2.Sales*
East	Boston	Boston	700 €
West	Los Angeles	Los Angeles	1500 €
West	Los Angeles	Los Angeles	300 €
West	San Diego	San Diego	250 €

- Para obtener la información de ventas por región, debemos combinar la información de las dos tablas. Al examinar las dos tablas, encontramos que están enlazadas a través del campo común “nombre_negocio” Primero presentaremos la instrucción SQL y explicaremos el uso de cada segmento después:

```
SELECT A1.region_name REGION, SUM(A2.Sales) SALES
FROM Geography A1, Store_Information A2
WHERE A1.store_name = A2.store_name
GROUP BY A1.region_name
```

Resultado:

REGIÓN	SALES
East	700 €
West	2050 €

1. Las primeras dos líneas le indican a SQL que seleccione dos campos, el primero es el campo "nombre_región" de la tabla ***Geography*** (denominado REGIÓN), y el segundo es la suma del campo "Sales" de la tabla ***Store_Information*** (denominado SALES). Note como se utilizan los alias de tabla aquí: Geografía se denomina A1, e Información_Negocio se denomina A2. Sin los alias, la primera línea sería

SELECT Geography.region_name REGION, SUM(Store_Information.Sales) SALES

que es mucho más problemática. En esencia, los alias de tabla facilitan el entendimiento de la totalidad de la instrucción SQL, especialmente cuando se incluyen tablas múltiples.

2. Luego, pongamos nuestra atención en la línea 2, la instrucción **WHERE**. Aquí es donde se especifica la condición de la unión. En este caso, queremos asegurarnos que el contenido en “nombre_negocio” en la tabla Geografía concuerde con la tabla ***Store_Information***, y la forma de hacerlo es igualarlos. Esta instrucción **WHERE** es esencial para asegurarse de que obtenga el resultado correcto. Sin la correcta instrucción **WHERE** se producirá una Unión Cartesiana.

Ejercicios:

1- Regiones en las que la compañía tiene sedes (sin repetir).

Nota: ¿En que tabla(s) está la información que se necesita? ¿Que información se necesita (pensar tanto en columnas como en relaciones entre datos?

2- Regiones en las que se ha realizado alguna venta el 8 de Enero de 1999.

Nota: ¿En que tabla(s) está la información que se necesita? ¿Que información se necesita (pensar tanto en columnas como en relaciones entre datos?

3- Media de ventas en cada región

4- Ventas en cada región ordenadas por región y ciudad

14. OUTER JOIN

- Anteriormente, hemos visto una unión natural o interna, donde seleccionamos filas comunes a las tablas que participan en la unión. ¿Qué sucede en los casos si estamos interesados en la selección de todos los elementos en una de las tablas sin importar si se encuentran presentes en la otra tabla? Ahora necesitaremos utilizar el comando **SQL OUTER JOIN**.

- La sintaxis para realizar una unión externa en SQL depende de la base de datos. Por ejemplo, en Oracle, **colocaremos un "(+)" en la cláusula WHERE del lado opuesto de la tabla para la que queremos incluir todas las filas.**

```
SELECT "nombrex_columna", "nombrey_columna"
FROM "nombre_tabla1"   “nombre_tabla2”
WHERE     “nombre_tabla1”.”nombre1_columna”
          = “nombre_tabla2”.”nombre2_columna” (+)
```

  Este esquema correspondería a un LEFT JOIN, porque estamos incluyendo todas las filas de la tabla que está a la izquierda de la condición (de igualdad) del join. Equivalentemente, se puede escribir un RIGHT JOIN, colocando '(+)' al otro lado.

- Digamos que tenemos las siguientes dos tablas:

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Tabla ***Geography***

region_name	store_name
East	Boston
East	New York
West	Los Angeles
West	San Diego
South	Lousiana

y queremos saber la suma de las ventas de todos los negocios. Si realizamos una unión regular, no podríamos obtener lo que deseamos debido a que habríamos omitido “Nueva York" ni “Louisiana” ya que no aparecen en la tabla ***Store_Information***. Por lo tanto, necesitamos realizar una unión externa respecto de las dos tablas anteriores:

```
SELECT A1.store_name, SUM(A2.Sales) SALES
FROM Geography A1, Store_Information A2
WHERE A1.store_name = A2.store_name (+)
GROUP BY A1.store_name
```

Note que en este caso, estamos utilizando la sintaxis Oracle para unión externa.

Resultado:

store_name	SALES
Boston	700 €
New York	
Los Angeles	1800 €
San Diego	250 €
Louisiana	

Nota: Se devuelve NULL cuando no hay coincidencia en la segunda tabla. En este caso, “Nueva York" y “Louisiana” no aparece en la tabla ***Store_Information***, por lo tanto su columna "SALES" correspondiente es NULL.

- Ejercicios:

1- Obtener la suma de ventas de cada región, en el caso de que se realizaran ventas en esa región, ordenando por región

2- Obtener un listado de las ventas se registraron en Enero de 1999 en cada región ordenado por región; en particular se desea que aparezcan las regiones en las que ese día no hubo ventas

3-Obtener la suma de ventas de cada región ordenando por región; como caso particular, aunque una región no tenga registrada ventas, debe aparecer en el listado esa región (sin ningún valor asociado de ventas). Resolver el ejercicio de dos formas diferentes.

4-Obtener la media de las ventas de cada región ordenando por región.
Nota: hay que definir lo que significa “media de ventas de una región”?

15. CONCAT (función sobre cadenas)

- Algunas veces es necesario combinar en forma conjunta (concatenar) los resultados de varios campos diferentes. Cada base de datos brinda una forma para realizar esto:

  - MySQL: CONCAT()
  - Oracle: CONCAT(), ||
  - SQL Server: +

  Nota: Las funciones que veremos a continuación (**CONCAT, SUBSTRING, TRIM**) NO son funciones de grupo; eso quiere decir que se aplican sobre datos individuales de una misma fila, no sobre grupos de datos

- La sintaxis para CONCAT() es la siguiente:

  CONCAT(cad1, cad2, cad3, ...): Concatenar cad1, cad2, cad3, y cualquier otra cadena juntas.

  Note que la función CONCAT() de Oracle sólo permite dos argumentos – sólo dos cadenas pueden colocarse juntas al mismo tiempo utilizando esta función. Sin embargo, es posible concatenar más de dos cadenas al mismo tiempo en Oracle utilizando '||'.

- Observemos algunos ejemplos. Supongamos que tenemos la siguiente tabla:

Tabla ***Geography***

region_name	store_name
East	Boston
East	New York
West	Los Angeles
West	San Diego

- Ejemplo 1:

  MySQL/Oracle:
  SELECT CONCAT(region_name,store_name) FROM Geography
  WHERE store_name = 'Boston';

  Resultado :

  'EastBoston'

- Ejemplo 2:

  Oracle:
  SELECT region_name || ' ' || store_name FROM Geography
  WHERE store_name = 'Boston';

  Resultado :

  'East Boston'

- Ejemplo 3:

  SQL Server:
  SELECT region_name + ' ' + store_name FROM Geography
  WHERE store_name = 'Boston';

  Resultado :

  'East Boston'

16. SUBSTRING (función)

- La función de subcadena en SQL se utiliza para tomar una parte de los datos almacenados. Esta función tiene diferentes nombres según las diferentes bases de datos:
  - MySQL: SUBSTR(), SUBSTRING()
  - Oracle: SUBSTR()
  - SQL Server: SUBSTRING()

- Los usos más frecuentes son los siguientes (utilizaremos SUBSTR() aquí):

  - **SUBSTR(str,pos)**: Selecciona todos los caracteres de <str> comenzando con posición <pos>. Note que esta sintaxis no es compatible en SQL Server.
  - **SUBSTR(str,pos,len)**: Comienza con el carácter <pos> en la cadena <str> y selecciona los siguientes caracteres <len>.

- Supongamos que tenemos la siguiente tabla:

Tabla Geography

region_name	store_name
East	Boston
East	New York
West	Los Angeles
West	San Diego

- Ejemplo 1 :

```
SELECT SUBSTR(store_name, 3)
FROM Geography
WHERE store_name = 'Los Angeles';
```

Resultado :

's Angeles'

- Ejemplo 2 :

```
SELECT SUBSTR(store_name,2,4)
FROM Geography
WHERE store_name = 'San Diego';
```

Resultado :

'an D'

17. TRIM (función)

- La función TRIM en SQL se utiliza para eliminar un prefijo o sufijo determinado de una cadena. El patrón más común a eliminarse son los espacios en blanco. Esta función tiene diferentes nombres según las diferentes bases de datos:
  - MySQL: TRIM(), RTRIM(), LTRIM()
  - Oracle: RTRIM(), LTRIM()
  - SQL Server: RTRIM(), LTRIM()

- La sintaxis para estas funciones de reducción es:

- **TRIM([[LOCATION] [remstr] FROM] str)**: [LOCATION] puede ser LÍDER, REMANENTE, o AMBAS. Esta función se deshace del patrón [remstr] tanto para el comienzo de la cadena como para el final, o para ambos. Si no se especifica ningún [remstr], los espacios en blanco se eliminarán.
- **LTRIM(str)**: Elimina todos los espacios en blanco del comienzo de la cadena.
- **RTRIM(str)**: Elimina todos los espacios en blanco del final de la cadena.

- Ejemplo:

  - Ejemplo 1 :

```
SELECT TRIM('   Sample   ');
```

Resultado :

```
'Sample'
```

  - Ejemplo 2 :

```
SELECT LTRIM('   Sample   ');
```

Resultado :

```
'Sample   '
```

  - Ejemplo 3 :

```
SELECT RTRIM('   Sample   ');
```

Resultado :

```
'   Sample'
```

II- SELECT (Conjuntos y subselects)

1. UNION

- El propósito del comando SQL **UNION** es combinar los resultados de dos consultas. **Una restricción de UNION es que ambos consultas debe proyectar sobre el mismo número de columnas y todas estas columnas deben ser del mismo tipo de datos**. El resultado de la consulta no obtiene valores repetidos.

  No confundir la **UNION** con Join, ya que aunque ambos permiten, en algún sentido, reunir datos, conceptualmente son operaciones diferentes:

  -La **UNION** es simplemente una unión de conjuntos en el sentido matemático del término; se combinan “filas”, todas ellas con el mismo número de columnas y del mismo tipo

  -Un Join es una operación que “combina” filas de dos tablas (según la condición de igualdad expresada en el Join sobre sendos campos de cada tabla que deben ser del mismo tipo), obteniendo “nuevas filas”, con “mas columnas” que las filas originales.

- La sintaxis es la siguiente:

  [Instrucción SQL 1]
  UNION
  [Instrucción SQL 2]

- Supongamos que tenemos las siguientes dos tablas,

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Tabla ***Internet_Sales***

Date	Sales
07-Jan-1999	250 €
10-Jan-1999	535 €
11-Jan-1999	320 €
12-Jan-1999	750 €

y deseamos saber de todas las fechas donde hay una operación de venta, independientemente de que haya sido una venta presencial o por Internet.

Para hacerlo, utilizamos la siguiente instrucción SQL:

```
SELECT Date FROM Store_Information
UNION
SELECT Date FROM Internet_Sales
```

Resultado:

Date
05-Jan-1999
07-Jan-1999
08-Jan-1999
10-Jan-1999
11-Jan-1999
12-Jan-1999

Note que si ingresamos "**SELECT DISTINCT Date**" para cada o ambas instrucciones SQL, obtendremos el mismo conjunto de resultados (debido a que la operación **UNION** elimina los duplicados).

2. UNION ALL

- El propósito del Comando SQL **UNION ALL** es también combinar los resultados de dos consultas juntas. La diferencia entre **UNION ALL** y **UNION** es que, mientras **UNION** sólo selecciona valores distintos, **UNION ALL** selecciona todos los valores.

- La sintaxis para **UNION ALL** es la siguiente:

```
[Instrucción SQL 1]
UNION ALL
[Instrucción SQL 2]
```

- Utilicemos el mismo ejemplo de la sección anterior para ilustrar la diferencia. Supongamos que tenemos las siguientes dos tablas,

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Tabla ***Internet_Sales***

Date	Sales
07-Jan-1999	250 €
10-Jan-1999	535 €
11-Jan-1999	320 €
12-Jan-1999	750 €

y deseamos encontrar las fechas en donde se realizó una operación de venta en un negocio como así también las fechas donde hay una venta a través de Internet. Para hacerlo, utilizamos la siguiente instrucción SQL:

```
SELECT Date FROM Store_Information
UNION ALL
SELECT Date FROM Internet_Sales
```

Resultado:

Date
05-Jan-1999
07-Jan-1999
08-Jan-1999
08-Jan-1999
07-Jan-1999
10-Jan-1999
11-Jan-1999
12-Jan-1999

3. INTERSECT

- Parecido al comando **UNION**, **INTERSECT** también opera en dos instrucciones SQL. La diferencia es que, mientras **UNION** actúa fundamentalmente como un operador **OR (O)** (el valor se selecciona si aparece en la primera o la segunda instrucción), el comando **INTERSECT** actúa como un operador **AND (Y)** (el valor se selecciona si aparece en ambas instrucciones).

- La sintaxis es la siguiente:

```
[Instrucción SQL 1]
INTERSECT
[Instrucción SQL 2]
```

- Digamos que tenemos las siguientes dos tablas:

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Tabla ***Internet_Sales***

Date	Sales
07-Jan-1999	250 €
10-Jan-1999	535 €
11-Jan-1999	320 €
12-Jan-1999	750 €

y deseamos encontrar todas las fechas donde hay ventas tanto en el negocio como en Internet. Para hacerlo, utilizamos la siguiente instrucción SQL:

```
SELECT Date FROM Store_Information
INTERSECT
SELECT Date FROM Internet_Sales
```

Resultado:

Date
07-Jan-1999

Note que el comando **INTERSECT** sólo arrojará valores distintivos.

4. MINUS

- **MINUS** opera en dos instrucciones SQL. Toma todos los resultados de la primera instrucción SQL, y luego sustrae aquellos que se encuentran presentes en la segunda instrucción SQL para obtener una respuesta final. Si la segunda instrucción SQL incluye resultados que no están presentes en la primera instrucción SQL, dichos resultados se ignoran.

- La sintaxis es la siguiente:

[Instrucción SQL 1]
MINUS
[Instrucción SQL 2]

- Continuemos con el mismo ejemplo:

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Tabla ***Internet_Sales***

Date	Sales
07-Jan-1999	250 €
10-Jan-1999	535 €
11-Jan-1999	320 €
12-Jan-1999	750 €

y deseamos encontrar todas las fechas donde hay ventas en el negocio, pero no aquellas realizadas por Internet.

Para hacerlo, utilizamos la siguiente instrucción SQL:

```
SELECT Date FROM Store_Information
MINUS
SELECT Date FROM Internet_Sales
```

Resultado:

Date
05-Jan-1999
08-Jan-1999

"05-Jan-1999", "07-Jan-1999", y "08-Jan-1999" son los valores distintivos obtenidos por "**SELECT Date FROM Store_Information**." También se obtiene "07-Jan-1999" de la segunda instrucción SQL, "**SELECT Date FROM Internet_Sales**," pero se lo excluye del conjunto final de resultados.

Note que el comando **MINUS** sólo arrojará valores distintos.

Algunas bases de datos pueden utilizar **EXCEPT** en vez de **MINUS**. Verifique la documentación para su base de datos específica para el uso apropiado.

5. Subconsultas

- Es posible incorporar una instrucción SQL dentro de otra. Cuando esto se hace en las instrucciones **WHERE** o **HAVING**, tenemos una construcción de subconsulta.

- La sintaxis es la siguiente:

```
SELECT "nombre1_columna"
FROM "nombre1_tabla"
WHERE "nombre2_columna"
        [Operador de Comparación]

        (SELECT "nombre3_columna"
        FROM "nombre2_tabla"
        WHERE [Condición])
```

[Operador de Comparación] podrían ser operadores de igualdad tales como **=, >, <, >=, <=**;También puede ser un operador textual como **LIKE**; la cláusula **(NOT) IN**; la cláusula **EXISTS**

La parte en **rojo** se considera como la "consulta interna", mientras que la parte en **verde** se considera como la "consulta externa".

- Utilizaremos el mismo ejemplo que empleamos para ilustrar los Join en SQL:

Table ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Table ***Geography***

region_name	store_name
East	Boston
East	New York
West	Los Angeles
West	San Diego

1) para encontrar las ventas de todas las ciudades en el oeste (West) usando una subconsulta, utilizamos la siguiente sentencia SQL:

```
SELECT SUM(Sales) FROM Store_Information
WHERE Store_name IN
(SELECT store_name FROM Geography
WHERE region_name = 'West')
```

Resultado:

SUM(Sales)
2050

En este ejemplo, en lugar de directamente combinar con un Join las dos tablas y sumar la cantidad de ventas de las ciudades en el Oeste (West), se utilizará en primer lugar la subconsulta para encontrar las ciudades en la zona oeste (West) y a continuación sumar la cantidad de ventas de estas tiendas.

2-) En el ejemplo anterior, la consulta interna se ejecuta en primer lugar, el resultado se envía a la consulta externa. Este tipo de subconsulta se llama **subconsulta simple**. Si la consulta interna depende de la consulta externa, vamos a tener una subconsulta correlacionada. A continuación encontrará un ejemplo de **subconsulta correlacionada**:

```
SELECT SUM(a1.Sales)
FROM Store_Information a1
WHERE a1.Store_name IN
                    (SELECT store_name
                    FROM Geography a2
                    WHERE a2.store_name = a1.store_name)
```

Fijese en la cláusula **WHERE** en la consulta interna, donde la condición requiere una referencia a la tabla de la consulta externa.

6. EXISTS

- En la sección anterior, utilizamos **IN** para enlazar la consulta interna y la consulta externa en una instrucción de subconsulta. **IN** no es la única forma de hacerlo. Se pueden utilizar otros operadores tales como **>**, **<**, o **=**. **EXISTS** es un operador especial que describiremos en esta sección.

  EXISTS simplemente verifica si la consulta interna arroja alguna fila. Si lo hace, entonces la consulta externa procede. De no hacerlo, la consulta externa no se ejecuta, y la totalidad de la instrucción SQL no arroja nada.

- La sintaxis para **EXISTS** es

```
SELECT "nombre1_columna"
FROM "nombre1_tabla"
WHERE EXISTS
        (SELECT *
        FROM "nombre2_tabla"
        WHERE [Condición])
```

  Note que en vez de *, puede seleccionar una o más columnas en la consulta interna. El efecto será idéntico.

- Utilizamos las mismas tablas de ejemplos:

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
Los Angeles	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

Tabla ***Geography***

region_name	store_name
East	Boston
East	New York
West	Los Angeles
West	San Diego

colocaríamos la siguiente consulta SQL:

```
SELECT SUM(Sales) FROM Store_Information
WHERE EXISTS
(SELECT * FROM Geography
WHERE region_name = 'West')
```

Obtendremos el siguiente resultado:

SUM(Sales)
2050

Al principio, esto puede parecer confuso, debido a que la subsequencia incluye la condición [region_name = 'West'], aún así la consulta sumó los negocios para todas las regiones. Si observamos de cerca, encontramos que debido a que la subconsulta arroja más de 0 filas, la condición **EXISTS** es verdadera, y la condición colocada dentro de la consulta interna no influye la forma en que se ejecuta la consulta externa.

7. CASE

- **CASE** se utiliza para brindar un tipo de lógica "si-entonces-otro" para SQL.

- Su sintaxis es:

```
SELECT CASE ("nombre_columna")
  WHEN "condición1" THEN "resultado1"
  WHEN "condición2" THEN "resultado2"
  ...
  [ELSE "resultadoN"]
  END
FROM "nombre_tabla"
```

"condición" puede ser un valor estático o una expresión. La cláusula **ELSE** es opcional.

- En nuestra Tabla ***Store_Information*** de ejemplo,

Tabla ***Store_Information***

store_name	Sales	Date
Los Angeles	1500 €	05-Jan-1999
San Diego	250 €	07-Jan-1999
San Francisco	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

si deseamos multiplicar las sumas de ventas de 'Los Angeles' por 2 y las sumas de ventas de 'San Diego' por 1,5, ingresamos,

```
SELECT store_name, CASE store_name
  WHEN 'Los Angeles' THEN Sales * 2
  WHEN 'San Diego' THEN Sales * 1.5
  ELSE Sales
  END
"Nuevas Ventas",
Date
FROM Store_Information
```

"Nuevas Ventas" es el alias que se le otorga a la columna con la instrucción CASE.

Resultado:

store_name	Nuevas Ventas	Date
Los Angeles	3000 €	05-Jan-1999
San Diego	375 €	07-Jan-1999
San Francisco	300 €	08-Jan-1999
Boston	700 €	08-Jan-1999

III- Resumen de la sintaxis de SQL

El propósito de esta página es brindar una página de referencia rápida para la sintaxis SQL.

■ SQL-DQL: SELECT

Select
SELECT "nom de columna" FROM "nombre_tabla"

Distinct
SELECT DISTINCT "nombre_columna"
FROM "nombre_tabla"

Where
SELECT "nombre_columna"
FROM "nombre_tabla"
WHERE "condition"

And/Or
SELECT "nombre_columna"
FROM "nombre_tabla"
WHERE "condición simple"
{[AND|OR] "condición simple"}+

In
SELECT "nombre_columna"
FROM "nombre_tabla"
WHERE "nombre_columna" IN ('valor1', 'valor2', ...)

Between
SELECT "nombre_columna"
FROM "nombre_tabla"
WHERE "nombre_columna" BETWEEN 'valor1' AND 'valor2'

Like
SELECT "nombre_columna"
FROM "nombre_tabla"
WHERE "nombre_columna" LIKE {patrón}

Order By
SELECT "nombre_columna"
FROM "nombre_tabla"
[WHERE "condición"]
ORDER BY "nombre_columna" [ASC, DESC]

Count
SELECT COUNT("nombre_columna")

FROM "nombre_tabla"

Group By
SELECT "nombre_columna 1", SUM("nombre_columna 2")
FROM "nombre_tabla"
GROUP BY "nombre_columna 1"

Having
SELECT "nombre_columna 1", SUM("nombre_columna 2")
FROM "nombre_tabla"
GROUP BY "nombre_columna 1"
HAVING (condición de función aritmética)

■ SQL-DDL

Create Table
CREATE TABLE "nombre_tabla"
("columna 1" "tipo_de_datos_para_columna_1",
"columna 2" "tipo_de_datos_para_columna_2",
...)

Drop Table
DROP TABLE "nombre_tabla"

Truncate Table
TRUNCATE TABLE "nombre_tabla"

Insert Into
INSERT INTO "nombre_tabla" ("colonne 1", "colonne 2", ...)
valorS ("valor 1", "valor 2", ...)

Update
UPDATE "nombre_tabla"
SET "colonne 1" = [nuevo valor]
WHERE {condition}

Delete From
DELETE FROM "nombre_tabla"
WHERE {condición}