SQL (Lenguaje de consulta estructurado)

Es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones sobre las mismas. Una de sus características es el manejo del álgebra y el cálculo relacional permitiendo lanzar consultas con el fin de recuperar -de una forma sencilla- información de interés de una base de datos, así como también hacer cambios sobre la misma. Es un lenguaje de cuarta generación (4GL).

Características generales

El SQL es un lenguaje de acceso a bases de datos que explota la flexibilidad y potencia de los sistemas relacionales permitiendo gran variedad de operaciones sobre los mismos.

Es un lenguaje declarativo de alto nivel o de no procedimiento, que gracias a su fuerte base teórica y su orientación al manejo de conjuntos de registros, y no a registros individuales, permite una alta productividad en codificación. De esta forma una sola sentencia puede equivaler a uno o más sentencias de código en un lenguaje de bajo nivel.

Optimización

Como ya se dijo arriba, y suele ser común en los lenguajes de acceso a bases de datos de alto nivel, el SQL es un lenguaje declarativo. O sea, que especifica qué es lo que se quiere y no cómo conseguirlo, por lo que una sentencia no establece explícitamente un orden de ejecución.

El orden de ejecución interno de una sentencia puede afectar gravemente a la eficiencia del SGBD, por lo que se hace necesario que éste lleve a cabo una optimización antes de la ejecución de la misma. Muchas veces, el uso de índices acelera una instrucción de consulta, pero ralentiza la actualización de los datos. Dependiendo del uso de la aplicación, se priorizará el acceso indexado o una rápida actualización de la información. La optimización difiere sensiblemente en cada motor de base de datos y depende de muchos factores.

Lenguaje de definición de datos (LDD)

El lenguaje de definición de datos (en inglés Data Definition Language, o DDL), es el que se encarga de la modificación de la estructura de los objetos de la base de datos. Existen cuatro operaciones básicas: CREATE, ALTER, DROP y TRUNCATE.

CREATE

Este comando crea un objeto dentro de la base de datos. Puede ser una tabla, vista, índice, trigger, función, procedimiento o cualquier otro objeto que el motor de la base de datos soporte.

Ejemplo (crear una tabla) [editar]

```
CREATE TABLE "TABLA_NOMBRE" (
"CAMPO_1" INT,
"CAMPO_2" STRING
)
```

ALTER

Este comando permite modificar la estructura de un objeto. Se pueden agregar/quitar campos a una tabla, modificar el tipo de un campo, agregar/quitar índices a una tabla, modificar un trigger, etc.

Ejemplo (agregar columna a una tabla) [editar]

```
ALTER TABLE "TABLA_NOMBRE" (
ADD NUEVO_CAMPO INT UNSIGNED
)
```

DROP

Este comando elimina un objeto de la base de datos.Puede ser una tabla, vista, índice, trigger, función, procedimiento o cualquier otro objeto que el motor de la base de datos soporte. Se puede combinar con la sentencia ALTER.

Ejemplo

```
ALTER TABLE "TABLA_NOMBRE"
(
DROP COLUMN "CAMPO_NOMBRE1"
)
```

TRUNCATE

Este comando trunca todo el contenido de una tabla. La ventaja sobre el comando DELETE, es que si se quiere borrar todo el contenido de la tabla, es mucho más rápido, especialmente si la tabla es muy grande, la desventaja es que TRUNCATE solo sirve cuando se quiere eliminar absolutamente todos los registros, ya que no se permite la cláusula WHERE. Si bien, en un principio, esta sentencia parecería ser DML (Lenguaje de Manipulación de Datos), es en realidad una DDL, ya que internamente, el comando truncate borra la tabla y la vuelve a crear y no ejecuta ninguna transacción.

Ejemplo

TRUNCATE TABLE "TABLA_NOMBRE"

Lenguaje de manipulación de datos (LMD)

Definición

Un lenguaje de manipulación de datos (Data Manipulation Language, o DML en inglés) es un lenguaje proporcionado por el sistema de gestión de base de datos que permite a los usuarios de la misma llevar a cabo las tareas de consulta o manipulación de los datos, organizados por el modelo de datos adecuado.

El lenguaje de manipulación de datos más popular hoy día es SQL, usado para recuperar y manipular datos en una base de datos relacional. Otros ejemplos de DML son los usados por bases de datos IMS/DL1, CODASYL u otras.

INSERT

Una sentencia INSERT de SQL agrega uno o más registros a una (y sólo una) tabla en una base de datos relacional.

Forma básica

INSERT INTO "tabla" ("columna1", ["columna2,... "]) VALUES ("valor1", ["valor2,..."])

Las cantidades de columnas y valores deben ser las mismas. Si una columna no se especifica, le será asignado el valor por omisión. Los valores

especificados (o implícitos) por la sentencia INSERT deberán satisfacer todas las restricciones aplicables. Si ocurre un error de sintaxis o si alguna de las restricciones es violada, no se agrega la fila y se devuelve un error.
Ejemplo
INSERT INTO agenda_telefonica (nombre, numero) VALUES ('Roberto Fernández', '4886850');
Cuando se especifican todos los valores de una tabla, se puede utilizar la sentencia acortada:
INSERT INTO "tabla" VALUES ("valor1", ["valor2,..."])
Ejemplo (asumiendo que 'nombre' y 'numero' son las únicas columnas de la tabla 'agenda_telefonica'):
INSERT INTO agenda_telefonica VALUES ('Roberto Fernández', '4886850');

Formas avanzadas
Inserciones en múltiples filas
Una característica de SQL (desde SQL-92) es el uso de constructores de filas para insertar múltiples filas a la vez, con una sola sentencia SQL:
INSERT INTO "tabla" ("columna1", ["columna2,... "])
VALUES ("valor1a", ["valor1b,..."]), ("value2a", ["value2b,..."]),...

INSERT INTO agenda_telefonica VALUES ('Roberto Fernández', '4886850'), ('Alejandro Sosa', '4556550');
Que podía haber sido realizado por las sentencias
INSERT INTO agenda_telefonica VALUES ('Roberto Fernández', '4886850');
INSERT INTO agenda_telefonica VALUES ('Alejandro Sosa', '4556550');

Notar que las sentencias separadas pueden tener semántica diferente (especialmente con respecto a los triggers), y puede tener diferente performance que la sentencia de inserción múltiple.

Para insertar varias filas en MS SQL puede utilizar esa construcción:
INSERT INTO phone_book

```
SELECT 'John Doe', '555-1212'
UNION ALL
SELECT 'Peter Doe', '555-2323';
```

Tenga en cuenta que no se trata de una sentencia SQL válida de acuerdo con el estándar SQL (SQL: 2003), debido a la cláusula subselect incompleta.

Un INSERT también puede utilizarse para recuperar datos de otros, modificarla si es necesario e insertarla directamente en la tabla. Todo esto se hace en una sola sentencia SQL que no implica ningún procesamiento intermedio en la aplicación cliente. Un SUBSELECT se utiliza en lugar de la cláusula VALUES. El SUBSELECT puede contener JOIN, llamadas a funciones, y puede incluso consultar en la misma TABLA los datos que se inserta. Lógicamente, el SELECT se evalúa antes que la operación INSERT este iniciada. Un ejemplo se da a continuación.

```
INSERT INTO phone_book2

SELECT *
FROM   phone_book
WHERE  name IN ('John Doe', 'Peter Doe')
```

Una variación es necesaria cuando algunos de los datos de la tabla fuente se esta insertando en la nueva tabla, pero no todo el registro. (O cuando los esquemas de las tablas no son lo mismo.)

```
INSERT INTO phone_book2 ( [name], [phoneNumber] )

SELECT [name], [phoneNumber]
FROM   phone_book
WHERE  name IN ('John Doe', 'Peter Doe')
```

El SELECT produce una tabla (temporal), y el esquema de la tabla temporal debe coincidir con el esquema de la tabla donde los datos son insertados.

UPDATE

Una sentencia UPDATE de SQL es utilizada para modificar los valores de un conjunto de registros existentes en una tabla.

Forma básica

UPDATE ''tabla'' SET ''columna1'' = ''valor1'' [,''columna2'' = valor2'',...]
WHERE ''columnaN = ''valorN''
Ejemplo

UPDATE My_table SET field1 = 'updated value' WHERE field2 = 'N';

DELETE
Una sentencia DELETE de SQL borra cero o más registros existentes en una tabla,
Forma básica

DELETE FROM ''tabla'' WHERE ''columna1'' = ''valor1''

Ejemplo
DELETE FROM My_table WHERE field2 = 'N';

Recuperación de clave
Los diseñadores de base de datos que usan una clave suplente como la clave principal para cada tabla, se ejecutará en el ocasional escenario en el que es necesario recuperar automáticamente la base de datos, generando una clave primaria de una sentencia SQL INSERT para su uso en otras sentencias SQL. La mayoría de los sistemas no permiten sentencias SQL INSERT para retornar fila de datos. Por lo tanto, se hace necesario aplicar una solución en tales escenarios.
Implementaciones comunes incluyen:
Utilizando un procedimiento almacenado especifico de base de datos que genera la clave suplente, realice la operación INSERT, y finalmente devuelve la clave generada.
Utilizando una sentencia SELECT específica de base de datos, sobre una tabla temporal que contiene la última fila insertada. DB2 implementa esta característica de la siguiente manera:

```
SELECT *
FROM   NEW TABLE ( INSERT INTO phone_book VALUES ( 'Peter Doe','555-2323' ) ) AS t
```

Utilizando una sentencia SELECT después de la sentencia INSERT con función específica de base de datos, que devuelve la clave primaria generada por el más reciente registro insertado.
Utilizando una combinación única de elementos de el original SQL INSERT en una posterior sentencia SELECT.

Desencadenantes

Si desencadenantes son definidos sobre la tabla en la que opera la sentencia INSERT, los desencadenantes son evaluados en el contexto de la operación. Desencadenantes BEFORE INSERT permiten la modificación de los valores que se insertará en la tabla. desencadenantes AFTER INSERT no puede modificar los datos de ahora en adelante, pero se puede utilizar para iniciar acciones en otras tablas, por ejemplo para aplicar mecanismos de auditoría.

La sentencia SELECT

La sentencia **SELECT** nos permite consultar los datos almacenados en una tabla de la base de datos.

El formato de la sentencia select es:

SELECT [**ALL** | **DISTINCT**]
<*nombre_campo*> [{,<*nombre_campo*>}]
FROM <*nombre_tabla*>|<*nombre_vista*>
[{,<*nombre_tabla*>|<*nombre_vista*>}]
[**WHERE** <*condicion*> [{ **AND**|**OR** <condicion>}]]
[**GROUP BY** <*nombre_campo*> [{,<*nombre_campo* >}]]
[**HAVING** <*condicion*>[{ **AND**|**OR** <condicion>}]]
[**ORDER BY** <*nombre_campo*>|<indice_campo> [**ASC** | **DESC**]
[{,<*nombre_campo*>|<indice_campo> [**ASC** | **DESC**]}]]

Veamos por partes que quiere decir cada una de las partes que conforman la sentecia.

	Significado
SELECT	Palabra clave que indica que la sentencia de SQL que queremos ejecutar es de selección.
ALL	Indica que queremos seleccionar todos los valores.Es el valor por defecto y no suele especificarse casi nunca.
DISTINCT	Indica que queremos seleccionar sólo los valores distintos.
FROM	Indica la tabla (o tablas) desde la que queremos recuperar los datos. En el caso de que exista más de una tabla se denomina a la consulta "consulta combinada" o "join". En las consultas combinadas es necesario aplicar una condición de combinación a través de una cláusula **WHERE**.
WHERE	Especifica una condición que debe cumplirse para que los datos sean devueltos por la consulta. Admite los operadores lógicos **AND** y **OR**.
GROUP BY	Especifica la agrupación que se da a los datos. Se usa siempre en combinación con funciones agregadas.
HAVING	Especifica una condición que debe cumplirse para los datosEspecifica una condición que debe cumplirse para que los datos sean devueltos por la consulta. Su funcionamiento es similar al de **WHERE** pero aplicado al conjunto de resultados devueltos por la consulta. Debe aplicarse siempre junto a **GROUP BY** y la condicion debe estar referida a los campos contenidos en ella.
ORDER BY	Presenta el resultado ordenado por las columnas indicadas. El orden puede expresarse con **ASC** (orden ascendente) y **DESC** (orden descendente). El valor predeterminado es **ASC.**

```
SELECT matricula,
       marca,
       modelo,
       color,
       numero_kilometros,
       num_plazas
FROM tCoches
ORDER BY marca,modelo;
```

La palabra clave **FROM** indica que los datos serán recuperados de la tabla tCoches.

Tambien podríamos haber simplicado la consulta a través del uso del comodin de campos, el asterisco "*".

```
SELECT *
FROM tCoches
ORDER BY marca,modelo;
```

El uso del asterisco indica que queremos que la consulta devuelva todos los campos que existen en la tabla.

La cláusula WHERE

La cláusula **WHERE** es la instrucción que nos permite filtrar el resultado de una sentencia **SELECT**. Habitualmente no deseamos obtener toda la información existente en la tabla, sino que queremos obtener sólo la información que nos resulte util es ese momento. La cláusula **WHERE** filtra los datos antes de ser devueltos por la consulta.

En nuestro ejemplo, si queremos consultar un coche en concreto debemos agregar una cláusula **WHERE**. Esta cláusula especifica una o varias condiciones que deben cumplirse para que la sentencia **SELECT** devuelva los datos. Por ejemplo, para que la consulta devuelva sólo los datos del coche con maricula M-1525-ZA debemos ejecutar la siguiente sentencia:

```
SELECT matricula,
       marca,
       modelo,
       color,
       numero_kilometros,
       num_plazas
FROM tCoches
WHERE matricula = 'M-1525-ZA';
```

Cuando en una cláusula where queremos incluir un tipo texto, debemos incluir el valor entre comillas simples.

Además, podemos utilizar tantas condiciones como queramos, utilizando los operadores lógicos **AND** y **OR** . El siguiente ejemplo muestra una consulta que devolverá los coches cuyas matriculas sean M-1525-ZA o bien M-2566-AA.

```
SELECT matricula,
       marca,
       modelo,
       color,
       numero_kilometros,
       num_plazas
FROM tCoches
WHERE matricula = 'M-1525-ZA'
   OR matricula = 'M-2566-AA' ;
```

Además una condición **WHERE** puede ser negada a través del operador lógico **NOT.** La siguiente consulta devolverá todos los datos de la tabla tCohes menos el que tenga matricula M-1525-ZA.

```
SELECT matricula,
        marca,
        modelo,
        color,
        numero_kilometros,
        num_plazas
FROM tCoches
WHERE NOT matricula = 'M-1525-ZA'  ;
```

Podemos tambien obtener las diferentes marcas y modelos de coches ejecutando la consulta.

```
SELECT DISTINCT marca,
                modelo
FROM tCoches;
```

La ver los valores distintos. En el caso anterior se devolveran lpalabra clave **DISTINCT** indica que sólo queremos os valores distintos del par formado por los campos marca y modelo.

La cláusula ORDER BY

Como ya hemos visto en los ejemplos anteriores podemos especificar el orden en el que serán devueltos los datos a través de la cláusula **ORDER BY**.

```
SELECT matricula,
        marca,
        modelo,
        color,
        numero_kilometros,
        num_plazas
FROM tCoches
ORDER BY marca ASC,modelo DESC;
```

Como podemos ver en el ejemplo podemos especificar la ordenación ascendente o descendente a través de las palabras clave **ASC** y **DESC**. La ordenación depende del tipo de datos que este definido en la columna, de forma que un campo número será ordenado como tal, y un alfanúmerico se ordenará de la A a la Z, aunque su contenido sea número. De esta forma el valor 100 se devuelve antes que el 11.

También podemos especificar el en la cláusula **ORDER BY** el índice número del campo dentro del la sentencia **SELECT** para la ordenación, el siguiente ejemplo ordenaría los datos por el campo marca, ya que aparece en segundo lugar dentro de la lista de campos que componen la **SELECT.**

```
SELECT matricula,
       marca,
       modelo,
       color,
       numero_kilometros,
       num_plazas
FROM tCoches
ORDER BY 2;
```

Combinación interna.

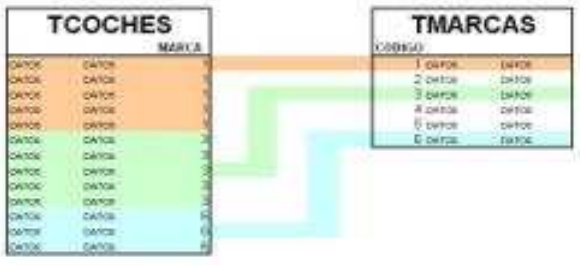
La combinación interna nos permite mostrar los datos de dos o más tablas a través de una condición **WHERE.**

Consulta **SELECT** en cuya claúsula **FROM** escribiremos el nombre de las dos tablas, separados por comas, y una condición **WHERE** que obligue a que el código de marca de la tabla de coches sea igual al código de la tabla de marcas.

Lo más sencillo es ver un ejemplo directamente:

```
SELECT tCoches.matricula,
              tMarcas.marca,
              tCoches.modelo,
              tCoches.color,
              tCoches.numero_kilometros,
              tCoches.num_plazas
FROM tCoches, tMarcas
WHERE tCoches.marca = tMarcas.codigo
```

La misma consulta de forma "visual" ...



Demonos cuenta que hemos antepuesto el nombre de cada tabla a el nombre del campo, esto no es obligatorio si los nombres de campos no se repiten en las tablas, pero es acondajable para evitar conflictos de nombres entre campos. Por ejemplo, si para referirnos al campo marca no anteponemos el nombre del campo la base de datos no sabe si queremos el campo marca de la tabla tCoches, que contiene el código de la marca, o el campo marca de la tabla tMarcas, que contiene el nombre de la marca.

Otra opción es utilizar la cláusula **INNER JOIN**. Su sintaxis es identica a la de una consulta **SELECT** habitual, con la particularidad de que én la cláusula **FROM** sólo aparece una tabla o vista, añadiendose el resto de tablas a través de cláusulas **INNER JOIN** .

```
SELECT [ALL | DISTINCT ]
            <nombre_campo> [{,<nombre_campo>}]
FROM <nombre_tabla>
[{INNER JOIN  <nombre_tabla> ON <condicion_combinacion>}]
[WHERE <condicion> [{ AND|OR <condicion>}]]
[GROUP BY <nombre_campo> [{,<nombre_campo >}]]
[HAVING <condicion>[{ AND|OR <condicion>}]]
[ORDER BY <nombre_campo>|<indice_campo> [ASC | DESC]
                 [{,<nombre_campo>|<indice_campo> [ASC | DESC ]}]]
```

El ejemplo anterior escrito utilizando la clausula **INNER JOIN** quedaria de la siguiente manera:

```
SELECT tCoches.matricula,
       tMarcas.marca,
       tCoches.modelo,
       tCoches.color,
       tCoches.numero_kilometros,
       tCoches.num_plazas
FROM tCoches
INNER JOIN tMarcas ON tCoches.marca = tMarcas.codigo
```

La cláusula **INNER JOIN** permite separar completamente las condiciones de combinación con otros criterios, cuando tenemos consultas que combinan nueve o diez tablas esto realmente se agradece. Sin embargo muchos programadores no son amigos de la cláusula **INNER JOIN**, la razón es que uno de los principales gestores de bases de datos, **ORACLE**, no la soportaba.

Combinación Externa

La combinación interna es excluyente. Esto quiere decir que si un registro no cumple la condición de combinación no se incluye en los resultados. De este modo en el ejemplo anterior si un coche no tiene grabada la marca no se devuelve en mi consulta.

Según la naturaleza de nuestra consulta esto puede ser una ventaja , pero en otros casos significa un serio problema. Para modificar este comportamiento SQL pone a nuestra disposición la combinación externa. La combinación externa no es excluyente.

La sintaxis es muy parecida a la combinación interna,

```
SELECT [ALL | DISTINCT ]
            <nombre_campo> [{,<nombre_campo>}]
FROM <nombre_tabla>
[{LEFT|RIGHT OUTER JOIN  <nombre_tabla> ON <condicion_combinacion>}]
[WHERE <condicion> [{ AND|OR <condicion>}]]
[GROUP BY <nombre_campo> [{,<nombre_campo >}]]
[HAVING <condicion>[{ AND|OR <condicion>}]]
[ORDER BY <nombre_campo>|<indice_campo> [ASC | DESC]
                 [{,<nombre_campo>|<indice_campo> [ASC | DESC ]}]]
```

La combinación externa puede ser diestra o siniestra, **LEFT OUTER JOIN** o **RIGHT OUTER JOIN.** Con **LEFT OUTER JOIN** obtenemos todos los registros de en la tabla que situemos a la izquierda de la clausula **JOIN,** mientras que con **RIGHT OUTER JOIN** obtenmos el efecto contrario.

Como mejor se ve la combinación externa es con un ejemplo.

```
SELECT tCoches.matricula,
       tMarcas.marca,
       tCoches.modelo,
       tCoches.color,
       tCoches.numero_kilometros,
       tCoches.num_plazas
FROM tCoches
LEFT OUTER JOIN tMarcas ON tCoches.marca = tMarcas.codigo
```

Esta consulta devolverá todos los registros de la tabla tCoches, independientemente de que tengan marca o no. En el caso de que el coche no tenga marca se devolverá el valor **null** para los campos de la tabla tMarcas.

Visualmente (la consulta devuelve los datos en azul) ...

El mismo ejemplo con RIGHT OUTER JOIN.

```
SELECT tCoches.matricula,
       tMarcas.marca,
       tCoches.modelo,
       tCoches.color,
       tCoches.numero_kilometros,
       tCoches.num_plazas
FROM tCoches
RIGHT OUTER JOIN tMarcas ON tCoches.marca = tMarcas.codigo
```

Esta consulta devolverá los registros de la tabla tCoches que tengan marca relacionada y todos los registros de la tabla tMarcas, tengan algún registro en tCoches o no. Visualmente (la consulta devuelve los datos en azul) ...

Union

La cláusula **UNION** permite unir dos o más conjuntos de resultados en uno detras del otro como si se tratase de una única tabla. De este modo podemos obtener los registros de mas de una tabla "unidos".

La sintaxis corresponde a la de varias **SELECT** unidas a través de **UNION**, como se muestra a continuación:

```
SELECT [ALL | DISTINCT ]
            <nombre_campo> [{,<nombre_campo>}]
FROM <nombre_tabla>
[{LEFT|RIGHT OUTER JOIN  <nombre_tabla> ON <condicion_combinacion>}]
[WHERE <condicion> [{ AND|OR <condicion>}]]
[GROUP BY <nombre_campo> [{,<nombre_campo >}]]
[HAVING <condicion>[{ AND|OR <condicion>}]]
{
UNION [ALL | DISTINCT ]
SELECT [ALL | DISTINCT ]
            <nombre_campo> [{,<nombre_campo>}]
FROM <nombre_tabla>
[{LEFT|RIGHT OUTER JOIN  <nombre_tabla> ON <condicion_combinacion>}]
[WHERE <condicion> [{ AND|OR <condicion>}]]
[GROUP BY <nombre_campo> [{,<nombre_campo >}]]
[HAVING <condicion>[{ AND|OR <condicion>}]]
}
[ORDER BY <nombre_campo>|<indice_campo> [ASC | DESC]
                 [{,<nombre_campo>|<indice_campo> [ASC | DESC ]}]]
```

Para utilizar la clausula **UNION** debemos cumplir una serie de normas.

- Las consultas a unir deben tener el mismo número campos, y además los campos deben ser del mismo tipo.
- Sólo puede haber una única clausula **ORDER BY** al final de la sentencia **SELECT**.

El siguiente ejemplo muestra el uso de **UNION**

```
SELECT tCoches.matricula,
       tMarcas.marca,
       tCoches.modelo,
       tCoches.color,
       tCoches.numero_kilometros,
       tCoches.num_plazas
FROM tCoches
INNER JOIN tMarcas ON tCoches.marca = tMarcas.codigo
UNION
SELECT tMotos.matricula,
       tMarcas.marca,
       tMotos.modelo,
       tMotos.color,
       tMotos.numero_kilometros,
       0
FROM tMotos
INNER JOIN tMarcas ON tMotos.marca = tMarcas.codigo;
```