EAE Business School

We make it happen

Programa:

Máster en Big Data

Asignatura:

Complemento Formativo SQL

Profesor:

Joseba López-Samaniego

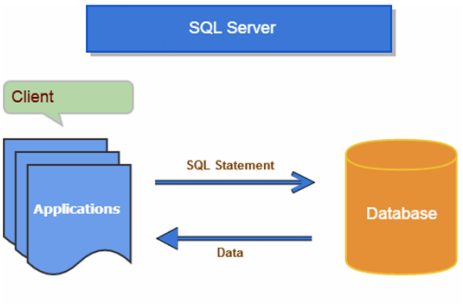
Campus / Convocatoria:

Madrid / Octubre 2020

CONTENIDO

TEMA 1: INTRODUCCIÓN A SQL SERVER

- Una base de datos es una entidad en la cual se pueden almacenar datos de manera estructurada, con la menor redundancia posible.
- Diferentes programas y diferentes usuarios deben poder utilizar estos datos. Por lo tanto, uno de los objetivos es el de poder compartir esta información.
- Rápidamente surgió la necesidad de contar con un sistema de administración para controlar tanto los datos como los usuarios. La administración de bases de datos se realiza con un Sistema Gestor de Bases de Datos (SGBD), como puede ser SQL Server.



OBJETIVOS DE LOS SGBD

Los principales objetivos de todo sistema gestor de base de datos son los siguientes:

- Definir la Base de Datos mediante el Lenguaje de Definición de Datos, el cual permite especificar la estructura, tipo de datos y las restricciones sobre los datos, almacenándolo todo en la propia base de datos.
- Separar la descripción y manipulación de los datos, permitiendo un mayor entendimiento de los objetos, además de flexibilidad de consulta y actualización.
- Permitir la inserción, eliminación, actualización, consulta de los datos mediante el Lenguaje de Manejo de Datos.
- Proporcionar acceso controlado a la base de datos.
  - Seguridad: los usuarios no autorizados no pueden acceder a la base de datos.
  - Integridad: mantiene la integridad y consistencia de la base de datos.
  - Control de Recurrencia: permite el acceso compartido a la base de datos.
  - Control de Recuperación: restablece la base de datos después de producirse un fallo de software o hardware.

- Diccionario de datos o Catálogo: contiene la descripción de los datos de la base de datos y es accesible por el usuario.

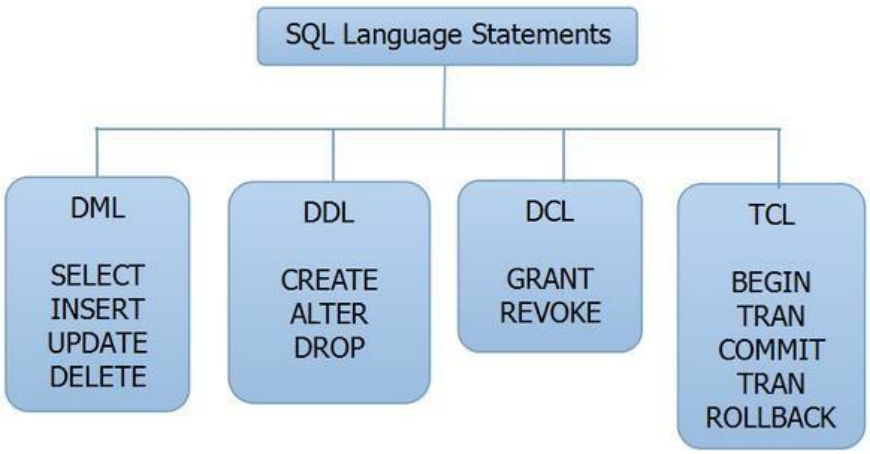
- Gestionar la estructura física de los datos y su almacenamiento, proporcionando eficiencia en las operaciones de la base de datos y el acceso al medio de almacenamiento.
- Proporcionar un mecanismo de vistas, que permita a cada usuario tener su propia vista o visión de la base de datos. El lenguaje de definición nos permite definir las vistas como subconjuntos de la base de datos, permitiendo:
  - Proporcionar un nivel de seguridad excluyendo datos para que no sean vistos por determinados usuarios.
  - Permiten que los usuarios vean los datos en el formato deseado.
  - Una vista representa una imagen consistente y permanente de la base de datos.
- Eliminar la redundancia de datos, establecer una mínima duplicidad y minimizar el espacio en disco utilizado.
- Proveer interfaces procedimentales y no procedimentales, permitiendo la manipulación por usuarios interactivos.
- Independizar la estructura de la organización lógica de los datos (Independencia física).
- Independizar la descripción lógica de la Base de datos y las descripciones particulares de los diferentes puntos de vistas de los usuarios.
- Permitir una fácil administración de los datos.

TEMA 2:LENGUAJE SQL

- Las Bases de Datos SQL Server se basan en el lenguaje SQL.
- SQL es un lenguaje estándar para acceder y manipular bases de datos relacionales.
- El lenguaje SQL está compuesto por comandos, cláusulas, operadores y funciones de agregado. Estos elementos se combinan en las instrucciones para crear, actualizar y manipular las bases de datos.

COMANDOS SQL

- DML (Data Manipulation Language): SELECT , INSERT, UPDATE, DELETE.
- DDL (Data Definition Language): ALTER, CREATE, DROP.
- DCL (Data Control Language): GRANT, REVOKE.
- TCL (Control de Transacción): BEGIN, TRAN, COMMIT, ROLLBACK.



CLÁUSULAS

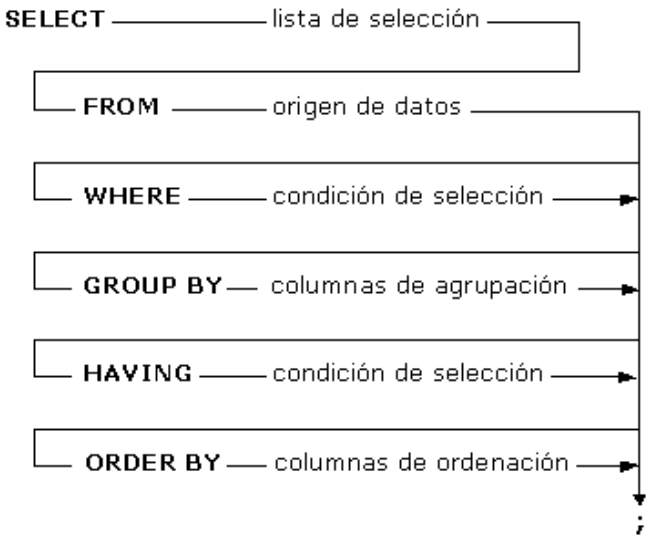
FROM: Enuncia la procedencia de los datos, indicando la TABLA , la VISTA o la SUBCONSULTA que se utilizará

WHERE: Se emplea para especificar el filtro o condición que se desea utilizar y que se debe cumplir

GROUP: Sirve para agrupar los registros de resultado y poder realizar funciones de agregado

ORDER: Otorga el orden a los resultados

HAVING: Se utiliza para dar la condición a cumplir por cada grupo, en caso de que exista agrupación de datos



OPERADORES

Operadores Lógicos

AND Es el “y” lógico. Evalúa dos condiciones y devuelve un valor de verdad sólo si ambas son ciertas.

OR Es el “o” lógico. Evalúa dos condiciones y devuelve un valor de verdad si alguna de las dos es cierta. NOTNegación lógica. Devuelve el valor contrario de la expresión.

Operadores de comparación

< Menor que

> Mayor que

<> Distinto de

<= Menor o igual que

>= Mayor o igual que

BETWEEN Intervalo

LIKE Comparación

In Especificar

FUNCIONES

Funciones de agregado

Las funciones de agregado se usan dentro de una cláusula SELECT para devolver un único valor que se aplica a un grupo de registros.

AVG Utilizada para calcular el promedio de los valores de un campo determinado
COUNTUtilizada para devolver el número de registros de la selección

SUM Utilizada para devolver la suma de todos los valores de un campo determinado
MAX Utilizada para devolver el valor más alto de un campo especificado

MIN Utilizada para devolver el valor más bajo de un campo especificado

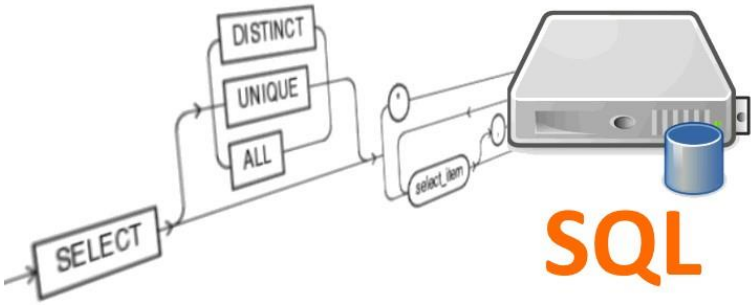
CONSULTAS DE SELECCIÓN

Las consultas de selección se utilizan para indicar al motor de datos que devuelva información de las bases de datos, esta información es devuelta en forma de conjunto de registros. Este conjunto de registros es modificable.

Para realizar consultas sobre las tablas de las bases de datos disponemos de la instrucción SELECT. Con ella podemos consultar una o varias tablas. Es sin duda el comando más versátil del lenguaje SQL.

Existen muchas cláusulas asociadas a la sentencia SELECT (GROUP BY, ORDER, HAVING, UNION).

Vamos a empezar viendo las consultas simples, basadas en una sola tabla. Veremos cómo obtener filas y columnas de una tabla en el orden en que nos haga falta.



El resultado de una consulta SELECT nos devuelve una tabla lógica. Es decir, los resultados son una relación de datos, que tiene filas/registros, con una serie de campos/columnas.

La sintaxis básica de una consulta SELECT es la siguiente (los valores opcionales van entre corchetes):

SELECT [ALL / DISTINC] [*] / [ListaColumnas_Expresiones] AS [Expresion]

FROM Nombre_Tabla_Vista

WHERE Condiciones

ORDER BY ListaColumnas [ASC / DESC]

SELECT Nombre, Telefono FROM Clientes;

Ordenar los registros

Se puede especificar el orden en que se desean recuperar los registros de las tablas mediante la clausula ORDER BY:

SELECT CodigoPostal, Nombre, Telefono FROM Clientes ORDER BY Nombre;

Se pueden ordenar los registros por mas de un campo:

SELECT CodigoPostal, Nombre, Telefono FROM Clientes ORDER BY CodigoPostal, Nombre;

Y se puede especificar el orden de los registros: ascendente mediante la clausula (ASC -se toma este valor por defecto) o descendente (DESC):

SELECT CodigoPostal, Nombre, Telefono FROM Clientes ORDER BY CodigoPostal DESC , Nombre ASC;

Consultas con predicado

ALL Si no se incluye ninguno de los predicados se asume ALL. El Motor de base de datos selecciona todos

los registros que cumplen las condiciones de la instrucción SQL:

SELECT ALL FROM Empleados;

SELECT * FROM Empleados;

TOP Devuelve un cierto número de registros que entran entre al principio o al final de un rango especificado por una cláusula ORDER BY. Supongamos que queremos recuperar los nombres de los 25 primeros estudiantes del curso 1994:

SELECT TOP 25 Nombre, Apellido FROM Estudiantes

ORDER BY Nota DESC;

Si no se incluye la cláusula ORDER BY, la consulta devolverá un conjunto arbitrario de 25 registros de la tabla Estudiantes .El predicado TOP no elige entre valores iguales. En el ejemplo anterior, si la nota media número 25 y la 26 son iguales, la consulta devolverá 26 registros. Se puede utilizar la palabra reservada PERCENT para devolver un cierto porcentaje de registros que caen al principio o al final de un rango especificado por la cláusula ORDER BY. Supongamos que en lugar de los 25 primeros estudiantes deseamos el 10 por ciento del curso:

SELECT TOP 10 PERCENT Nombre, Apellido FROM Estudiantes

ORDER BY Nota DESC;

DISTINCT Omite los registros que contienen datos duplicados en los campos seleccionados. Para que los valores de cada campo listado en la instrucción SELECT se incluyan en la consulta deben ser únicos:

SELECT DISTINCT Apellido FROM Empleados;

DISTINCTROW Devuelve los registros diferentes de una tabla; a diferencia del predicado anterior que sólo se fijaba en el contenido de los campos seleccionados, éste lo hace en el contenido del registro completo independientemente de los campo indicados en la cláusula SELECT:

SELECT DISTINCTROW Apellido FROM Empleados;

CRITERIOS DE SELECCIÓN

Operadores Lógicos

Algunos operadores lógicos soportados por SQL son: AND, OR, y Not.

Sintaxis:

<expresión1> operador <expresión2>…

En donde expresión1 y expresión2 son las condiciones a evaluar, el resultado de la operación varía en función del operador lógico:

```
# SELECT * FROM Empleados WHERE Edad > 25 AND Edad < 50;
# SELECT * FROM Empleados WHERE (Edad > 25 AND Edad < 50) OR Sueldo = 100;
# SELECT * FROM Empleados WHERE NOT Estado = 'Soltero';
# SELECT * FROM Empleados WHERE (Sueldo > 100 AND Sueldo < 500) OR (Provincia = 'Madrid' AND Estado = 'Casado');
```

Son expresiones lógicas a comprobar para la condición de filtro, que tras su resolución devuelven para cada fila TRUE o FALSE, en función de que se cumplan o no. Se puede utilizar cualquier expresión lógica y en ella utilizar otros operadores de comparación como:

- > (Mayor)
- >= (Mayor o igual)
- < (Menor)
- <= (Menor o igual)
- = (Igual)
- <> o != (Distinto)
- IS [NOT] NULL (para comprobar si el valor de una columna es o no es nula, es decir, si contiene o no contiene algún valor)

Se dice que una columna de una fila es NULL si está completamente vacía. Hay que tener en cuenta que si se ha introducido cualquier dato, incluso en un campo

alfanumérico si se introduce una cadena en blanco o un cero en un campo numérico, deja de ser NULL.

Operador BETWEEN

Para indicar que deseamos recuperar los registros según el intervalo de valores de un campo emplearemos el operador Between:

SELECT * FROM Pedidos WHERE CodPostal Between 28000 And 28999;

(Devuelve los pedidos realizados en la provincia de Madrid)

SELECT IIf(CodPostal Between 28000 And 28999, 'Provincial', 'Nacional') FROM Editores;

(Devuelve el valor 'Provincial' si el código postal se encuentra en el intervalo, 'Nacional' en caso contrario)

Operador LIKE

Se utiliza para comparar una expresión de cadena con un modelo en una expresión SQL.

Para ello utiliza los caracteres comodín especiales: "%" y "_". Con el primero indicamos que en su lugar puede ir cualquier cadena de caracteres, y con el segundo que puede ir cualquier carácter individual (un solo carácter). Con la combinación de estos caracteres podremos obtener múltiples patrones de búsqueda.

Su sintaxis es: expresión LIKE modelo

Ejemplos patrones de búsqueda:

- El nombre empieza por A: Nombre LIKE 'A%'
- El nombre acaba por A: Nombre LIKE '%A'
- El nombre contiene la letra A: Nombre LIKE '%A%'
- El nombre empieza por A y después contiene un solo carácter cualquiera: Nombre LIKE 'A_'
- El nombre empieza una A, después cualquier carácter, luego una E y al final cualquier cadena de caracteres: Nombre LIKE 'A_E%'

Operador IN

Este operador devuelve aquellos registros cuyo campo indicado coincide con alguno de los indicados en una lista. Su sintaxis es:

expresión [Not] In(valor1, valor2, . . .)

```
# SELECT * FROM Pedidos WHERE Provincia In ('Madrid', 'Barcelona', 'Sevilla');
```

Clausula WHERE

La cláusula WHERE puede usarse para determinar qué registros de las tablas enumeradas en la cláusula FROM aparecerán en los resultados de la instrucción SELECT. WHERE es opcional, pero cuando aparece debe ir a continuación de FROM:

```
# SELECT Apellidos, Salario FROM Empleados WHERE Salario > 21000;

# SELECT Id_Producto, Existencias FROM Productos WHERE Existencias <= Nuevo_Pedido;
```

Resumiendo, tal y como hemos visto es posible combinar varias condiciones simples de los operadores anteriores utilizando los operadores lógicos **AND, OR** y **NOT**, así como el uso de paréntesis para controlar la prioridad de los operadores (como en matemáticas). Por ejemplo: … (Cliente = 100 AND Provincia = 30) OR Ventas > 1000 … que sería para los clientes de las provincias 100 y 30 o cualquier cliente cuyas ventas superen 1000.

AGRUPAMIENTO DE REGISTROS (AGREGACIÓN)

AVG

Calcula la media aritmética de un conjunto de valores contenidos en un campo especificado de una consulta:

```
Avg(expr)
```

La función Avg no incluye ningún campo Null en el cálculo. Un ejemplo del funcionamiento de AVG:

```
# SELECT Avg(Gastos) AS Promedio FROM Pedidos WHERE Gastos > 100;
```

MAX, MIN

Devuelven el mínimo o el máximo de un conjunto de valores contenidos en un campo especifico de una consulta. Su sintaxis es:

```
Min(expr) Max(expr)
```

Un ejemplo de su uso:

```
# SELECT Min(Gastos) AS ElMin FROM Pedidos

WHERE Pais = 'Costa Rica';

# SELECT Max(Gastos) AS ElMax FROM Pedidos WHERE Pais = 'Costa Rica';
```

SUM

Devuelve la suma del conjunto de valores contenido en un campo especifico de una consulta. Su sintaxis es:

```
Sum(expr)
```

Por ejemplo:

```
# SELECT Sum(PrecioUnidad * Cantidad) AS Total FROM DetallePedido;
```

GROUP BY

Combina los registros con valores idénticos, en la lista de campos especificados, en un único registro:

```
# SELECT campos FROM tabla WHERE criterio GROUP BY campos del grupo
```

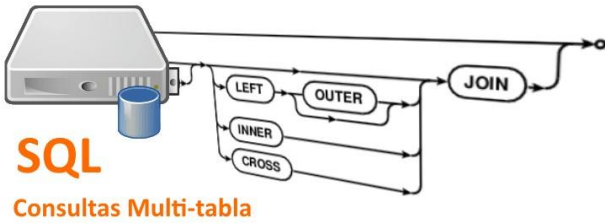
Todos los campos de la lista de campos de SELECT deben o bien incluirse en la cláusula GROUP BY o como argumentos de una función SQL agregada:

```
# SELECT Id_Familia, Sum(Stock)

FROM Productos GROUP BY Id_Familia;
```

HAVING es similar a WHERE, determina qué registros se seleccionan. Una vez que los registros se han agrupado utilizando GROUP BY, HAVING determina cuales de ellos se van a mostrar.

```
# SELECT Id_Familia Sum(Stock) FROM Productos

GROUP BY Id_Familia HAVING Sum(Stock) > 100 AND NombreProducto Like
BOS*;
```

CONSULTAS MULTI-TABLA



Es habitual que queramos acceder a datos que se encuentran en más de una tabla y mostrar información mezclada de todas ellas como resultado de una consulta. Para ello tendremos que hacer combinaciones de columnas de tablas diferentes.

En SQL es posible hacer esto especificando más de una tabla en la cláusula FROM de la instrucción SELECT.

Tenemos varias formas de obtener esta información.

Una de ellas consiste en crear combinaciones que permiten mostrar columnas de diferentes tablas como si fuese una sola tabla, haciendo coincidir los valores de las columnas relacionadas.

Este último punto es muy importante, ya que si seleccionamos varias tablas y no hacemos coincidir los valores de las columnas relacionadas, obtendremos una gran duplicidad de filas, realizándose el **producto cartesiano** entre las filas de las diferentes tablas seleccionadas.

Vamos a ver este importante detalle con un ejemplo simple. Consideremos estas tres consultas sobre la base de datos:

```
select count(*) from stars; select count(*) from planets;

select count(*) from stars, planets;
```

La primera instrucción devuelve 25000 filas (las estrellas de la galaxia), la segunda 31057 filas (los planetas), y la tercera 776425000 (que son 25000 x 31057, es decir, la combinación de todas las filas de estrellas y de planetas).

La otra manera de mostrar información de varias tablas -mucho más habitual y lógica- es uniendo filas de ambas, para ello es necesario que las columnas que se van a unir entre las dos tablas sean las mismas y contengan los mismos tipos de datos, es decir, mediante una clave externa.

```
select count(*) from stars, planets where stars.starid=planets.starid;
```

OPERACIONES DE UNIÓN - JOIN

La operación JOIN o combinación permite mostrar columnas de varias tablas como si se tratase de una sola tabla, combinando entre sí los registros relacionados usando para ello claves externas.

Las tablas relacionadas se especifican en la cláusula FROM, y además hay que hacer coincidir los valores que relacionan las columnas de las tablas.

Veamos un ejemplo, que selecciona el nombre de estrellas y el nombre de planetas de cada estrella:

```
SELECT s.name, p.name FROM planets p, stars s WHERE p.starid=s.starid
```

Para evitar que se produzca como resultado el producto cartesiano entre las dos tablas, expresamos el vínculo que se establece entre las dos tablas en la cláusula WHERE. En este caso relacionamos ambas tablas mediante el identificador de la estrella, clave existente en ambas. Fíjate en como le hemos otorgado un alias a cada tabla (p y s respectivamente) para no tener que escribir su nombre completo cada vez que necesitamos usarlas.

Hay que tener en cuenta que si el nombre de una columna existe en más de una de las tablas indicadas en la cláusula FROM, hay que poner, obligatoriamente, el nombre o alias de la tabla de la que queremos obtener dicho valor. En caso contrario nos dará un error de ejecución, indicando que hay un nombre ambiguo.

Hay otra forma adicional, que es más explícita y clara a la hora de realizar este tipo de combinaciones -y que se incorpora a partir de ANSI SQL-92- que permite utilizar una nueva cláusula llamada JOIN en la cláusula FROM, cuya sintaxis es el siguiente:

```
SELECT [ ALL / DISTINC ] [ * ] / [ListaColumnas_Expresiones]

FROM NombreTabla1 JOIN NombreTabla2 ON Condiciones_Vinculos_Tablas
```

De esta manera relacionamos de manera explícita ambas tablas sin necesidad de involucrar la clave externa en las condiciones del SELECT (o sea, en el WHERE). Es una manera más clara y limpia de llevar a cabo la relación.

Esto se puede ir aplicando a cuantas tablas necesitemos combinar en nuestras consultas. Veamos como quedaría el ejemplo anterior con el uso de JOIN:

```
SELECT s.name, p.name FROM planets p

 JOIN stars s ON p.starid=s.starid
```

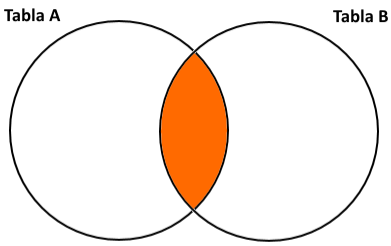
TIPOS DE JOIN

A parte del tipo de unión más básico que ya hemos visto, existen más formas de realizar la unión de tablas, que nos permitirán controlar mejor los conjuntos de resultados que obtenemos.

Combinaciones internas - INNER JOIN

Las combinaciones internas se realizan mediante la instrucción INNER JOIN. Devuelven únicamente aquellos registros/filas que tienen valores idénticos en los dos campos que se comparan para unir ambas tablas. Es decir aquellas que tienen elementos en las dos tablas, identificados éstos por el campo de relación.

La mejor forma de verlo es con un diagrama de Venn que ilustre en qué parte de la relación deben existir registros:



INNER JOIN

En este caso se devuelven los registros que tienen nexo de unión en ambas tablas. Por ejemplo, en la relación entre las tablas de estrellas y planetas en GalaXQL, se devolverán los registros de todos los planetas que orbiten alrededor de una estrella, relacionándolos por el ID de la estrella.

Esto puede ocasionar la desaparición del resultado de filas de alguna de las dos tablas, por tener valores nulos, o por tener un valor que no exista en la otra tabla entre los campos/columnas que se están comparando.

Su sintaxis es:

```
FROM Tabla1 [INNER] JOIN Tabla2 ON Condiciones_Vinculos_Tablas
```

Así, para seleccionar los registros comunes entre la Tabla1 y la Tabla2 que tengan correspondencia entre ambas tablas por el campo Col1, escribiríamos:

```
SELECT T1.Col1, T1.Col2, T1.Col3, T2.Col7

    FROM Tabla1 T1 INNER JOIN Tabla2 T2 ON T1.Col1 = T2.Col1
```

En realidad esto ya lo conocíamos puesto que en las combinaciones internas, el uso de la palabra **INNER** es opcional. Si simplemente indicamos la palabra JOIN y la combinación de columnas, el sistema sobreentiende que estamos haciendo una combinación interna.

Combinaciones externas (OUTER JOIN)

Las combinaciones externas se realizan mediante la instrucción OUTER JOIN. Como enseguida veremos, devuelven todos los valores de la tabla que hemos puesto a la derecha, los de la tabla que hemos puesto a la izquierda o los de ambas tablas según el caso, devolviendo además valores nulos en las columnas de las tablas que no tengan el valor existente en la otra tabla.

Es decir, que nos permite seleccionar algunas filas de una tabla aunque éstas no tengan correspondencia con las filas de la otra tabla con la que se combina. Ahora lo veremos mejor en cada caso concreto, ilustrándolo con un diagrama para una mejor comprensión.

La sintaxis general de las combinaciones externas es:

```
FROM Tabla1 [LEFT/RIGHT/FULL] [OUTER] JOIN Tabla2 ON
Condiciones_Vinculos_Tablas
```

Como vemos existen tres variantes de las combinaciones externas.

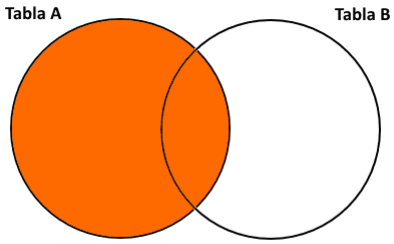
En todas estas combinaciones externas el uso de la palabra OUTER es opcional. Si utilizamos LEFT, RIGHT o FULL y la combinación de columnas, el sistema sobreentiende que estamos haciendo una combinación externa.

Variante LEFT JOIN

Se obtienen todas las filas de la tabla colocada a la izquierda, aunque no tengan correspondencia en la tabla de la derecha.

Así, para seleccionar todas las filas de la Tabla1, aunque no tengan correspondencia con las filas de la Tabla2, suponiendo que se combinan por la columna Col1 de ambas tablas escribiríamos:

```
SELECT T1.Col1, T1.Col2, T1.Col3, T2.Col7

FROM Tabla1 T1 LEFT [OUTER] JOIN Tabla2 T2 ON T1.Col1 = T2.Col1
```



LEFT OUTER JOIN

De este modo, volviendo a GalaXQL, si escribimos la siguiente consulta:

```
SELECT *

FROM planets p

LEFT JOIN moons m ON p.planetid=m.planetid
```

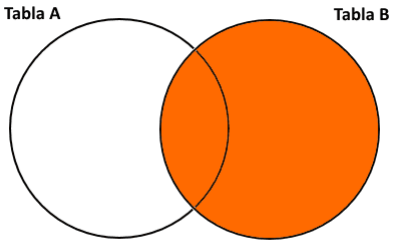
Obtenemos todos los planetas de la galaxia incluso los que no tengan lunas en su órbita.

Variante RIGHT JOIN

Análogamente, usando RIGHT JOIN se obtienen todas las filas de la tabla de la derecha, aunque no tengan correspondencia en la tabla de la izquierda.

Así, para seleccionar todas las filas de la Tabla2, aunque no tengan correspondencia con las filas de la Tabla1 podemos utilizar la cláusula RIGHT:

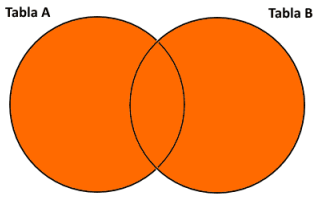
```
SELECT T1.Col1, T1.Col2, T1.Col3, T2.Col7

FROM Tabla1 T1 RIGHT [OUTER] JOIN Tabla2 T2 ON T1.Col1 = T2.Col1
```



RIGHT OUTER JOIN

Variante FULL JOIN

Se obtienen todas las filas en ambas tablas, aunque no tengan correspondencia en la otra tabla. Es decir, todos los registros de A y de B aunque no haya correspondencia entre ellos, rellenando con nulos los campos que falten:



FULL OUTER JOIN

Su sintaxis es:

```
SELECT T1.Col1, T1.Col2, T1.Col3, T2.Col7

FROM Tabla1 T1 FULL [OUTER] JOIN Tabla2 T2 ON T1.Col1 = T2.Col1
```

UNION vs UNION ALL

El operador UNION permite combinar los resultados de varias instrucciones SELECT en un único conjunto de resultados. Todos los conjuntos de resultados combinados mediante UNION deben tener la misma estructura. Deben tener el mismo número de columnas y las columnas del conjunto de resultados deben tener tipos de datos compatibles.

UNION no nos muestra los resultados duplicados, pero UNION ALL si los muestra: Con una tabla con datos: 1,2,3,4,5,6,7

Otra tabal con datos: 2,3,8,9,0

El resultado con Union seria: 1,2,3,4,5,6,7,8,9,0

El resultado con Union ALL seria: 1,2,2,3,3,4,5,6,7,8,9,0

Para resultados en los que tenemos claro que no se van a repetir los valores es conveniente usar Union ALL, debido a que Union hace un distinct de los datos, penalizando el rendimiento.

VISTAS

Las vistas ("views") en SQL son un mecanismo que permite generar un resultado a partir de una consulta (query) almacenado, y ejecutar nuevas consultas sobre este resultado como si fuera una tabla normal. Las vistas tienen la misma estructura que una tabla: filas y columnas. La única diferencia es que sólo se almacena de ellas la definición, no los datos.

La cláusula CREATE VIEW permite la creación de vistas. La cláusula asigna un nombre a la vista y permite especificar la consulta que la define. Su sintaxis es:

```
# CREATE VIEW id_vista [(columna,...)]AS especificación_consulta;
```

Opcionalmente se puede asignar un nombre a cada columna de la vista. Si se especifica, la lista de nombres de las columnas debe de tener el mismo número de elementos que el número de columnas producidas por la consulta. Si se omiten, cada columna de la vista1 adopta el nombre de la columna correspondiente en la consulta.

SUBCONSULTAS

Una subconsulta es una consulta anidada en una instrucción SELECT, INSERT, UPDATE o DELETE, o bien en otra subconsulta. Las subconsultas se pueden utilizar en cualquier parte en la que se permita una expresión.

Una subconsulta se denomina también consulta o selección interna, mientras que la instrucción que contiene la subconsulta es conocida como consulta o selección externa.

Aparece siempre encerrada entre paréntesis y tiene la misma sintaxis que una sentencia SELECT.

Se puede disponer de varios niveles de anidamiento, aunque el límite varía dependiendo de la memoria disponible y de la complejidad del resto de las expresiones de la consulta. Hay que tener en cuenta que para cada fila de la consulta externa, se calcula la subconsulta, si anidamos varias consultas, el número de veces que se ejecutarán las subconsultas, puede dispararse.

Cuando la subconsulta aparece en la lista de selección de otra consulta, deberá devolver un solo valor, de lo contrario provocará un error. Para reproducirlo: select *, (select * from planets) as planets from moons

Ejemplo de subconsulta: (obtener las lunas con radio más grande al planeta que orbitan)

```
select * from moons where radius > (select max(radius) from planets where
planetid=planetid)
```

Por cada fila de la tabla de lunas (de la consulta externa) se calcula la subconsulta y se evalúa la condición, por lo que utilizar una subconsulta puede en algunos casos 'ralentizar' la consulta, en contrapartida se necesita menos memoria que una composición de tablas.

Tenemos cuatro tipos de subconsultas:

- Las que devuelven un solo valor, aparecen en la lista de selección de la consulta externa o con un operador de comparación sin modificar.
  - Select * from planets where color = (select min(color) from planets)
  - En este caso, el gestor de la base de datos primero busca el código de color más pequeño, y luego devuelve todos los planetas que tengan el mismo color anteriormente buscado.

- Las que generan una columna de valores, aparecen con el operador (NOT) IN o con un operador de comparación modificado con ANY, SOME o ALL.
  - Útil para determinar si una expresión se halla incluida en los resultados de una sentencia SELECT.
  - select * from moons where color in (Select distinct color from planets)
  - En este caso, primero se buscan todos los colores diferentes que aparecen en la tabla planets y después se obtienen las lunas con los mismos colores que los planetas.
  - Si quisiéramos consultar las lunas con colores no iguales a los planetas solo tendríamos que modificar el operador por NOT IN. (select * from moons where color not in (Select distinct color from planets)
- Las que pueden generar cualquier número de columnas y filas, son utilizadas en pruebas de existencia especificadas con (NOT) EXISTS.
  - select * from moons m where exists (select * from planets p where p.planetid=m.planetid and p.radius=100)
  - En este ejemplo buscamos las lunas aspociadas a un planeta con radio = 100.
- Las que pueden usarse directamente como tablas. En caso de usarse en una JOIN se debe indicar un ALIAS y no duplicar nombres de columnas.

Aunque con subconsultas podemos estructurar mejor las sentencias, por su coste elevado, muchas veces es recomendable la misma resolución mediante JOIN's.

TABLAS TEMPORALES

- En el mundo de las bases de datos es muy común la utilización de tablas temporales.
- Puede ser útil recurrir a ellas porque muchas veces facilitan la resolución de problemas:
  - Almacenar datos para usarlos posteriormente.
  - Guardar resultados parciales.
  - Analizar grandes cantidades de filas.

Aunque la mayoría de veces puede suplirse con el uso de subconsultas.

ACTUALIZACIÓN Y DEFINICIÓN DE DATOS

INSERT

Hasta ahora hemos visto cómo extraer información de una base de datos. Sin embargo para poder extraer información de un almacén de datos, antes lógicamente debemos introducirla. En los sistemas gestores de datos relacionales la sentencia que nos permite hacerlo es INSERT.

La instrucción INSERT de SQL permite añadir registros a una tabla. Con ella podemos ir añadiendo registros uno a uno, o añadir de golpe tantos registros como nos devuelva una instrucción SELECT.

La sintaxis genérica de INSERT para crear un nuevo registro es la siguiente:

INSERT INTO NombreTabla [(Campo1, …, CampoN)] VALUES (Valor1, …, ValorN)

Siendo:

- NombreTabla: la tabla en la que se van a insertar las filas.
- (Campo1, …, CampoN): representa el campo o campos en los que vamos a introducir valores.
- (Valor1, …, ValorN): representan los valores que se van a almacenar en cada campo.

En realidad la lista de campos es opcional especificarla (por eso la hemos puesto entre corchetes en la sintaxis general). Si no se indica campo alguno se considera que por defecto vamos a introducir información en todos los campos de la tabla, y por lo tanto se deben introducir valores para todos ellos y en el orden en el que han sido definidos.

En realidad en la mayor parte de los casos los campos de identificación de los registros, también conocidos como claves primarias, serán campos de tipo numérico con auto-incremento. Esto quiere decir que el identificador único de cada registro (el campo que normalmente contiene un ID en su nombre) lo genera de manera automática el gestor de datos, incrementando en 1 su valor cada vez que se inserta un nuevo registro en la tabla. Por ello en estos casos no es necesario especificar este tipo de campos en las instrucciones INSERT.

Una segunda variante genérica de la instrucción INSERT es la que nos permite insertar de golpe múltiples registros en una tabla, bebiendo sus datos desde otra tabla (o varias tablas) de nuestra base de datos. En cualquier caso obteniéndolos a partir de una consulta SELECT convencional.

La sintaxis genérica de la instrucción que nos permite insertar registros procedentes de una consulta SELECT es la siguiente:

INSERT INTO NombreTabla [(Campo1, …, CampoN)] SELECT ...

El SELECT se indica a continuación de a lista de campos, y en lugar de especificar los valores con VALUES, se indica una consulta de selección. Es indispensable que los campos devueltos por esta instrucción SELECT coincidan en número y tipo de datos con los campos que se han indicado antes, o se producirá un error de ejecución y no se insertará ningún registro.

UPDATE

Una vez que hayamos introducido la información, casi seguro que tarde o temprano tendremos que actualizarla: un cliente cambia de dirección, se modifica la cantidad de un pedido, un empleado cambia de categoría... Todos estos sucesos implican actualizar información en nuestro modelo de datos. Para ayudarnos con eso, en SQL tenemos la instrucción UPDATE.

Esta instrucción nos permite actualizar los valores de los campos de una tabla, para uno o varios registros, o incluso para todos los registros de una tabla.

Su sintaxis general es:

UPDATE NombreTabla

SET Campo1 = Valor1, ..., CampoN = ValorN WHERE Condición

Siendo:

- NombreTabla: el nombre de la tabla en la que vamos a actualizar los datos.
- SET: indica los campos que se van a actualizar y con qué valores lo vamos a hacer.
- WHERE: Selecciona los registros de la tabla que se van a actualizar. Se puede aplicar todo lo visto para esta cláusula anteriormente, incluidas las sub-consultas.

DELETE

En los dos apartados anteriores hemos visto cómo realizar las operaciones de inserción de datos y

de actualización de esas inserciones. Ahora vamos a ver cómo eliminar datos una vez dejen de sernos

útiles. Para ello el estándar SQL nos ofrece la instrucción DELETE.

La instrucción DELETE permite eliminar uno o múltiples registros. Incluso todos los registros de una tabla, dejándola vacía.

Su sintaxis es general es: DELETE [FROM] NombreTabla WHERE Condición

La condición, como siempre, define las condiciones que deben cumplir los registros que se desean eliminar. Se puede aplicar todo lo visto para esta cláusula anteriormente, incluidas las sub-consultas.

IMPORTANTE: Al igual que hemos visto para las instrucciones de actualización, es extremadamente importante definir bien la cláusula WHERE. De otro modo podríamos eliminar muchos registros que no pretendíamos o incluso, en un caso extremo, borrar todas las filas de la tabla.

USO DE TRANSACCIONES

Entre las habilidades de todo Sistema Gestor de Bases de Datos tiene que estar la de permitir crear transacciones.

Una transacción es un conjunto de operaciones que van a ser tratadas como una sola.

Este conjunto de operaciones debe marcarse como transacción para que todas las operaciones que la conforman tengan éxito o todas fracasen.

La sentencia que se utiliza para indicar el comienzo de una transacción es 'BEGIN TRAN'. Si alguna de las operaciones de una transacción falla hay que deshacer la transacción en su totalidad para volver al estado inicial en el que estaba la base de datos antes de empezar. Esto se consigue con la sentencia 'ROLLBACK'.

Si todas las operaciones de una transacción se completan con éxito hay que marcar el fin de una transacción para que la base de datos vuelva a estar en un estado consistente con la sentencia 'COMMIT'.

```
BEGIN TRAN
    UPDATE Products SET UnitPrice=20 WHERE ProductName='Chang'
    UPDATE Products SET UnitPrice=20 WHERE ProductName='Chang'
COMMIT
```

Estas dos sentencias se ejecutarán como una sola. Si por ejemplo en medio de la transacción (después del primer update y antes del segundo) hay un corte de

conexión, cuando el SQL Server se recupere se encontrará en medio de una transacción y, o bien la termina o bien la deshace, pero no se quedará a medias.

TEMA 3: BASE DE DATOS RELACIONAL

- La base de datos relacional es un tipo de base de datos que cumple con el modelo relacional llevando implícitas las siguientes características:

  - Una base de datos se compone de varias tablas o relaciones.
  - No pueden existir dos tablas con el mismo nombre ni registro.
  - Cada tabla es a su vez un conjunto de campos (columnas) y registros (filas).
  - La relación entre una tabla padre y un hijo se lleva a cabo por medio de las claves primarias y claves foráneas.
  - Las claves primarias son la clave principal de un registro dentro de una tabla y estas deben cumplir con la integridad de datos.
  - Las claves foráneas se colocan en la tabla hija, contienen el mismo valor que la clave primaria del registro padre; por medio de estas se hacen las formas relacionales.

RESTRICCIONES

Las restricciones son utilizadas para asegurar la integridad de los datos almacenados en nuestras tablas.

Una restricción no deja de ser una limitación que obliga el cumplimiento de ciertas condiciones en la Base de Datos.

Algunas no son determinadas por los usuarios, sino que son inherentemente definidas por el simple hecho de que la BD sea relacional.

Los diferentes tipos de restricción que existen son:

- PRIMARY KEY
- UNIQUE
- FOREIGN KEY
- CHECK
- DEFAULT

PRIMARY KEY

Es la más común de todas debido a que cada una de nuestras tablas debe ser completamente relacional y para lograr esto siempre debe existir una clave primaria dentro de cada tabla que identifique cada fila como única.

Es posible agregar más columnas como parte de una clave primaria. Cada vez que generamos una clave primaria, esta crea un índice tipo clustered automáticamente.

Existen ciertos requerimientos para la creación de una clave primaria:

- La o las columnas utilizadas en una restricción PRIMARY KEY, no pueden aceptar NULL.
- No se pueden repetir valores en la o las columnas, deben ser únicos.
- Solamente puede existir una restricción de tipo PRIMARY KEY por cada tabla.

```
CREATE TABLE nombreEsquema.nombreTabla ( nombreColumna1 INT NOT
    NULL, nombreColumna2 VARCHAR(100) NOT NULL,

    CONSTRAINT PK_nombreRestriccion PRIMARY KEY( nombreColumna1
));
```

UNIQUE

Este tipo de restricción es muy parecida a PRIMARY KEY, las diferencias son las siguientes:

- También genera un índice automáticamente pero es de tipo NON CLUSTERED.
- La tabla puede tener más de una restricción de tipo UNIQUE.
- Si puede aceptar NULL, pero solo una fila puede contenerlo ya que como su nombre lo indica, es de tipo UNIQUE o único.

```
CREATE TABLE nombreEsquema.nombreTabla(
    nombreColumna1 INT NULL, nombreColumna2 VARCHAR(100) NOT NULL,

    CONSTRAINT UQ_nombreRestriccion UNIQUE( nombreColumna1 ),
    CONSTRAINT UQ_nombreRestriccion2 UNIQUE( nombreColumna2 ),

    CONSTRAINT UQ_nombreRestriccion3 UNIQUE(
    nombreColumna1,nombreColumna2 ));
```

FOREIGN KEY

Una clave foránea es una referencia a una clave en otra tabla, determina la relación existente en dos tablas. Las claves foráneas no necesitan ser claves únicas en la tabla donde están y sí a donde están referenciadas.

Por ejemplo, el código de departamento puede ser una clave foránea en la tabla de empleados. Se permite que haya varios empleados en un mismo departamento, pero habrá uno y sólo un departamento por cada clave de la tabla de departamentos.

Las columnas involucradas en la clave foránea deben tener el mismo tipo de datos que la clave primaria de la segunda tabla. Una clave foránea no crea un índice automáticamente, por lo que se recomienda generar uno para incrementar el rendimiento de la consulta.

Resumiendo requerimientos para la restricción FOREIGN KEY:

- Los valores ingresados en la o las columnas de la clave foránea, deben existir en la tabla a la que se hace referencia en la o las columnas de la clave primaria.
- Solo se pueden hacer referencia a claves primaria de tablas que se encuentren

dentro de la misma base de datos.

- Puede hacer referencia a otra columnas de la misma tabla.
- Solo puede hacer referencia a columnas de restricciones PRIMARY KEY o UNIQUE.
- No se puede utilizar en tablas temporales.

```
CREATE TABLE nombreEsquema.nombreTabla (
        nombreColumna1 INT NOT NULL, nombreColumna2 VARCHAR(100) NOT NULL,
    CONSTRAINT PK_nombreRestriccion PRIMARY KEY( nombreColumna1),
    CONSTRAINT FK_nombreRestriccion FOREIGN KEY( nombreColumna2) REFERENCES nombreEsquema.nombreTabla2 (nombreColumnaPKTabla2));
```

CHECK

Con este tipo de restricción, se especifica que los valores ingresados en la columna deben cumplir la regla o formula especificada. Por ejemplo:

```
CREATE TABLE nombreEsquema.nombreTabla(
    nombreColumna1 INT NULL,
    CONSTRAINT CH_nombreRestriccion CHECK (nombreColumna1>=0)); -- VALORES POSITIVOS
```

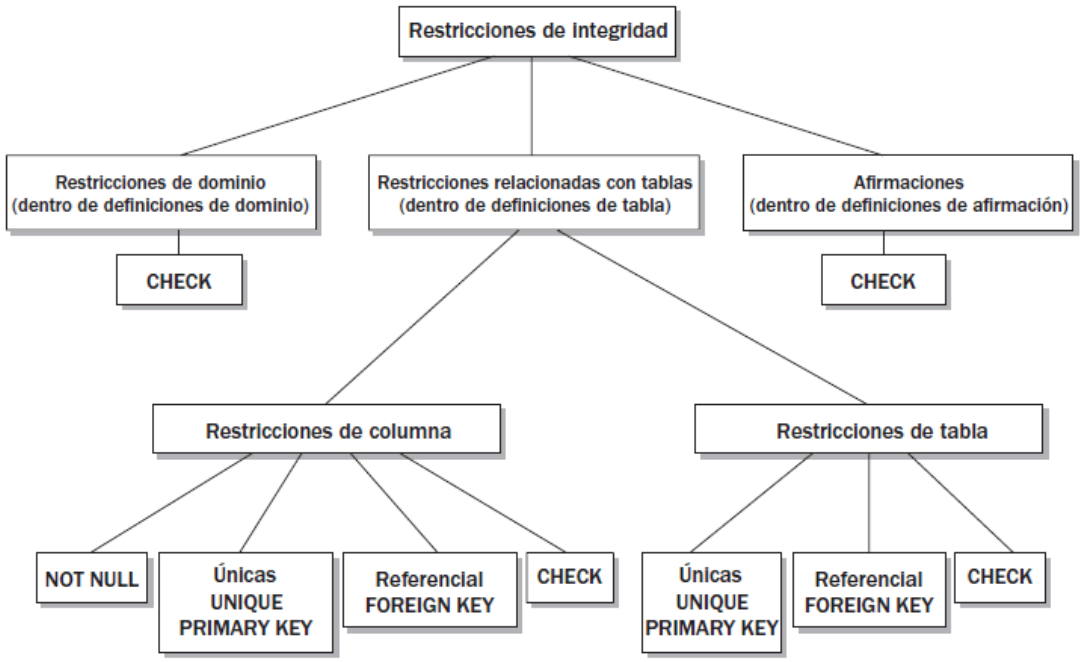
Algunos requerimientos son:

- Una columna puede tener cualquier número de restricciones CHECK.
- La condición de búsqueda debe evaluarse como una expresión booleana y no puede hacer referencia a otra tabla.
- No se pueden definir restricciones CHECK en columnas de tipo text, ntext o image.
- Al momento de crear nuestra expresión, tomar en cuenta si la columna acepta valores NULL, por ejemplo si definimos nuestra restricción que acepte solo valores positivos (nombreColumna1>=0), NULL es un valor desconocido por lo tanto se insertará en la columna.

DEFAULT

Se puede decir que no es una restricción, ya que solo se ingresa un valor en caso de que ninguno otro sea especificado. Si una columna permite **NULL** y el valor a insertar no se especifica, se puede sustituir con un valor predeterminado.

CREATE TABLE nombreTabla(nombreColumna1 INT NULL CONSTRAINT DF_nombreRestriccion DEFAULT(0));

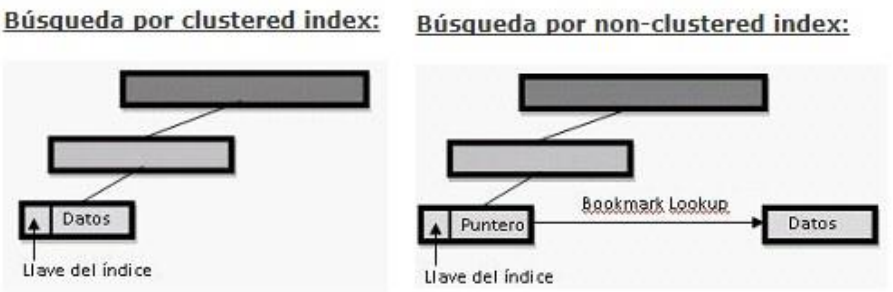


Aclaraciones sobre las restricciones

En todos los ejemplos se han configurado restricciones al crear tablas pero también se pueden añadir o modificar en tablas existentes utilizando el comando ALTER TABLE.

También hemos hablado de algunos conceptos que ahora vamos a describir:

- Índices: surgen con la necesidad de tener un acceso más rápido a los datos. Los índices pueden ser creados con cualquier combinación de campos de una tabla. Las consultas que filtran registros por medio de estos campos, pueden encontrar los registros de forma no secuencial usando la clave índice.
- Índice clustered (agrupados): Los índices clúster ordenan y almacenan las filas de los datos de la tabla o vista de acuerdo con los valores de la clave del índice. Son columnas incluidas en la definición del índice. Solo puede haber un índice clúster por cada tabla, porque las filas de datos solo pueden estar ordenadas de una forma.
- Índice non clustered (no agrupados): Los índices no clúster tienen una estructura separada de las filas de datos. Un índice no clúster contiene los valores de clave de índice no clúster y cada entrada de valor de clave tiene un puntero a la fila de datos que contiene el valor clave.



EJEMPLO MODELO RELACIONAL

ID_Localidad	ID_Prov	Nombre
03517	03	Abdet
03649	03	Manya
03729	03	Lliber
...		

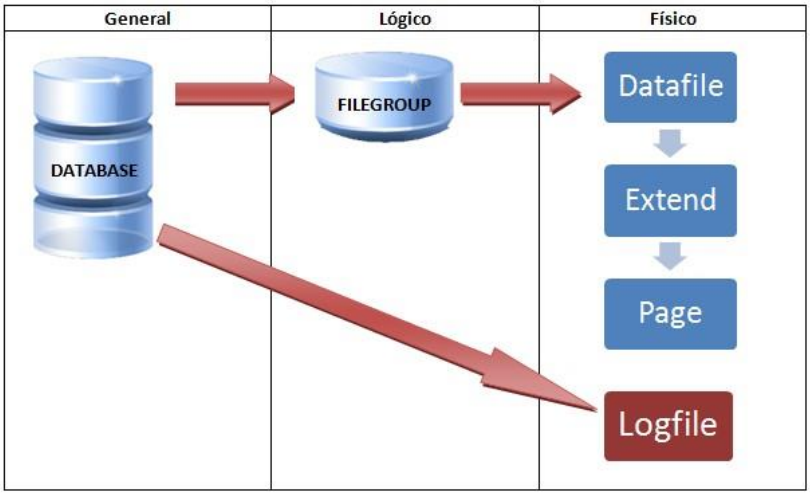
ID_Prov	Nombre
01	Álava (Vitoria-Gasteiz)
02	Albacete
03	Alicante
04	Almería
...	

ID_Emplead	Nombre	Apellidos	ID_Localidad	Telefono
1	Javi	Pérez Zamora	03517	678887765
2	Juan	Campos Villar	03517	654342342
3	Eusebio	Martínez Mir	03729	690435267

ESTRUCTURA INTERNA DE UNA BASE DE DATOS

En la siguiente tabla podemos ver cómo se organiza la información en una base de datos SQL Server:

Archivo	Descripción
Principal (Datafile)	El archivo de datos principal incluye la información de inicio de la base de datos y apunta a los demás archivos de la misma. Los datos y objetos del usuario se pueden almacenar en este archivo o en archivos de datos secundarios. Cada base de datos tiene un archivo de datos principal. La extensión recomendada para los nombres de archivos de datos principales es .mdf.
Secundario	Los archivos de datos secundarios son opcionales, están definidos por el usuario y almacenan los datos del usuario. Se pueden utilizar para distribuir datos en varios discos colocando cada archivo en una unidad de disco distinta. Además, si una base de datos supera el tamaño máximo establecido para un archivo de Windows, puede utilizar los archivos de datos secundarios para permitir el crecimiento de la base de datos. La extensión de nombre de archivo recomendada para los archivos de datos secundarios es .ndf.
Registro de transaccion es (Logfile)	Los archivos del registro de transacciones contienen la información de registro que se utiliza para recuperar la base de datos. Cada base de datos debe tener al menos un archivo de registro. La extensión recomendada para los nombres de archivos de registro es .ldf.


General
Lógico
Físico
DATABASE
FILEGROUP
Datafile
Extend
Page
Logfile

BASES DE DATOS DEL SISTEMA

SQL Server incluye las siguientes bases de datos del sistema.

Base de datos del sistema	Descripción
master	Registra toda la información del sistema para una instancia de SQL Server.
msdb	La utiliza el Agente SQL Server para programar alertas y trabajos.
model	Se utiliza como plantilla para todas las bases de datos creadas en la instancia de SQL Server. Las modificaciones hechas a la base de datos model , como el tamaño de la base de datos, la intercalación, el modelo de recuperación y otras opciones de base de datos, se aplicarán a las bases de datos que se creen con posterioridad.
tempdb	Área de trabajo que contiene objetos temporales o conjuntos de resultados intermedios.

TEMA 4: MODELADO DE DATOS

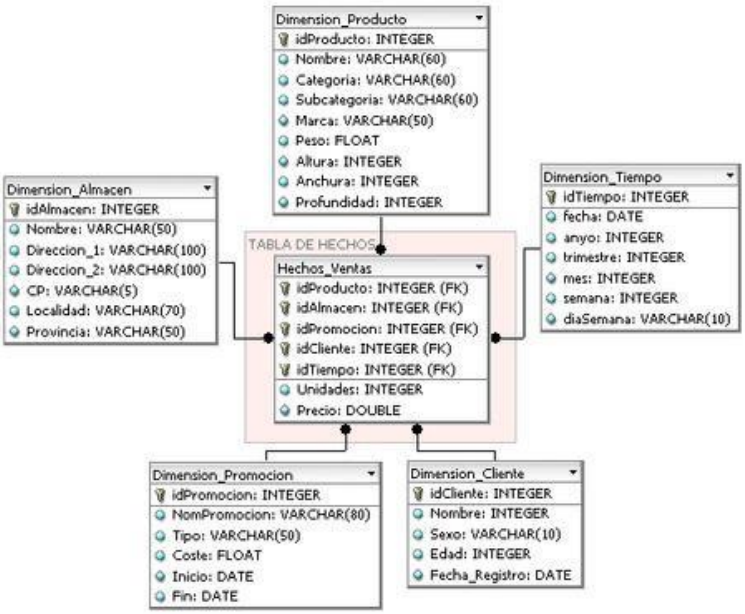
- Un modelo de base de datos es un tipo de modelo de datos que determina la estructura lógica de una base de datos y de manera fundamental determina el modo de almacenar, organizar y manipular los datos.
- Algunas de las cualidades que debe reflejar un buen modelo son las siguientes:
  - Reflejar la estructura del problema en el mundo real.
  - Ser capaz de representar todos los datos esperados, incluso con el paso del tiempo.
  - Evitar el almacenamiento de información redundante.
  - Proporcionar un acceso eficaz a los datos.
  - Mantener la integridad de los datos a lo largo del tiempo.
  - Ser claro, coherente y de fácil comprensión.
- Destacaremos los conceptos básicos de dos de las estructuras principales en el modelado de datos:
  - Esquema en estrella
  - Esquema en copo de nieve

ESQUEMA EN ESTRELLA

Un esquema en estrella es un modelo de datos que tiene una tabla de hechos (o tabla fact) que contiene los datos para el análisis, rodeada de las tablas de dimensiones. Este aspecto, de tabla de hechos (o central) más grande rodeada de radios o tablas más pequeñas es lo que asemeja a una estrella, dándole nombre a este tipo de construcciones.

Las tablas de dimensiones tendrán siempre una clave primaria simple, mientras que en la tabla de hechos, la clave principal estará compuesta por las claves principales de las tablas dimensionales.

El esquema estrella separa los datos del proceso de negocios en: hechos y dimensiones. Los hechos contienen datos medibles, cuantitativos, relacionados a la transacción del negocio, y las dimensiones son atributos que describen los datos indicados en los hechos (una especie de meta- datos, o sea datos que describen otros datos).



ESQUEMA EN COPO DE NIEVE

Un esquema en copo de nieve es una estructura algo más compleja que el esquema en estrella. Se da cuando alguna de las dimensiones se implementa con más de una tabla de datos. La finalidad es normalizar las tablas y así reducir el espacio de almacenamiento al eliminar la redundancia de datos; pero tiene la contrapartida de generar peores rendimientos al tener que crear más tablas de dimensiones y más relaciones entre las tablas (JOINS) lo que tiene un impacto directo sobre el rendimiento.

Al diseñar las tablas en las que se han de almacenar estos datos y parámetros, si se aplican las técnicas de normalización de bases de datos para optimizar el espacio requerido para guardar estos datos eliminando las redundancias, es habitual que se termine obteniendo un esquema en copo de nieve; en este tipo de esquemas se tiene una tabla central de hechos en la que se guardan las medidas del negocio que se quiere analizar, y en las tablas adyacentes se tendrán las dimensiones (parámetros) de que dependen los datos del negocio. Si por alguna dimensión se requiere más de una tabla (jerarquía de dimensiones) se dice que el esquema resultante es un esquema en copo de nieve.