Héroe en SQL: manual de iniciación

Válido para SQL Server, MySQL, Oracle y las bases de datos más populares

José Manuel Alarcón Aguín - CampusMVP

campus MVP

Los mejores cursos online para programadores
www.campusMVP.es

Sobre el autor

José Manuel Alarcón Aguín

José Manuel es ingeniero industrial y especialista en consultoría de empresa. Es socio fundador de campusMVP, empresa que dirige en la actualidad. Es también un reconocido profesional técnico, especializado en desarrollo web desde los inicios de la Web, y al que Microsoft ha reconocido en 10 ocasiones como Most Valuable Professional (MVP). Es autor de varios libros y cientos de artículos en revistas del sector como PC World, o Windows TI Magazine.

Twitter: @jm_alarcon

Blog: jasoft.org

¿Qué vas a encontrar?

1. ¿Qué es el lenguaje SQL? 4

2. Diseñando una base de datos en el modelo relacional 6

3. VÍDEO: Cómo instalar paso a paso la base de datos Northwind 14

4. Cómo realizar consultas simples con SELECT 16

5. Consultas SELECT multi-tabla - JOIN 25

6. Consultas SELECT multi-tabla - Tipos de JOIN 30

7. Operaciones con conjuntos 38

8. Agrupaciones y funciones de agregación 46

9. Funciones escalares en consultas de selección 54

10. Inserción de datos - INSERT 59

11. Actualización de datos - UPDATE 63

12. Eliminación de datos - DELETE 68

13. Transacciones 71

Acceso a datos

¿Qué es el lenguaje SQL?

El **Structured Query Language** o SQL es el lenguaje utilizado por la mayoría de los **Sistemas Gestores de Bases de Datos Relacionales** (SGBDR) surgidos a finales de los años 70, y que llega hasta nuestros días.

> A pesar del título de este capítulo, es una reiteración hablar de "Lenguaje SQL", ya que lo de "lenguaje" va ya en el nombre, aunque lo cierto es que casi todo el mundo lo dice así.

En 1986 fue estandarizado por el organismo ANSI *(American nacional Standard Institute),* dando lugar a la primera versión estándar de este lenguaje, el **SQL-86** o SQL1. Al año siguiente este estándar es adoptado también por el organismo internacional ISO *(International Standarization Organization)*.

> La parte fundamental de SQL es un estándar internacional.

A lo largo del tiempo se ha ido ampliando y mejorando. En la actualidad SQL es el estándar de *facto* de la inmensa mayoría de los SGBDR comerciales. El soporte es general y muy amplio, pero cada sistema (Oracle, SQL Server, MySQL...) incluye sus ampliaciones y pequeñas particularidades.

El ANSI SQL ha ido sufriendo varias revisiones a lo largo del tiempo, a continuación vienen indicadas en la siguiente tabla:

Año	Nombre	Alias	Comentarios
1986	SQL-86	SQL-87	Primera versión estándar de ANSI
1989	SQL-89	FIPS 127-1	Revisión menor, añade restricciones de integridad
1992	SQL-92	SQL2, FIPS 127-2	Actualización grande equivalente a ISO 9075
1999	SQL:1999	SQL3	Se añaden coincidencias mediante expresiones regulares, consultas recursivas, clausuras transitivas, disparadores, estructuras de control de flujo, tipos no escalares y algunas características de orientación a objetos.
2003	SQL:2003	SQL 2003	Se añaden características relacionadas con XML (la moda de la época), secuencias estandarizadas, y campos con valores auto-generados (incluyendo columnas de identidad).
2006	SQL:2006	SQL 2006	La ISO/IEC 9075-14:2006 define formas en las que SQL se puede utilizar con XML (importar y almacenar XML, manipularlo directamente en la base de datos...). Muy centrada en este aspecto concreto.
2008	SQL:2008	SQL 2008	Permite la cláusula ORDER BY fuera de las definiciones de cursores. Añade disparadores de tipo INSTEAD OF. Añade la cláusula TRUNCATE.[38]
2011	SQL:2011	SQL 2011	Añade características de manejo de datos temporales, así como tablas versionadas en el sistema, etc...

El lenguaje SQL se divide en tres subconjuntos de instrucciones, según la funcionalidad de éstas:

- **DML** (Data Manipulation Language – Lenguaje de Manipulación de Datos): se encarga de la manipulación de los datos. Es lo que usamos de manera más habitual para consultar, generar o actualizar información.
- **DDL** (Data Definition Language – Lenguaje de Definición de Datos): se encarga de la manipulación de los objetos de la base de datos, por ejemplo, crear tablas u otros objetos.
- **DCL** (Data Control Language – Lenguaje de Control de Datos): se encarga de controlar el acceso a los objetos y a los datos, para que los datos sean consistentes y sólo puedan ser accedidos por quien esté autorizado a ello.

Acceso a datos

Diseñando una base de datos en el modelo relacional

El diseño de una base de datos consiste en definir la estructura de los datos que debe tener un sistema de información determinado. Para ello se suelen seguir por regla general unas fases en el proceso de diseño, definiendo para ello el **modelo conceptual**, el **lógico** y el **físico.**

- En el **diseño conceptual** se hace una **descripción de alto nivel de la estructura de la base de datos,** independientemente del SGBD (Sistema Gestor de Bases de Datos) que se vaya a utilizar para manipularla. Su objetivo es describir el contenido de información de la base de datos y no las estructuras de almacenamiento que se necesitarán para manejar dicha información.

- El **diseño lógico** parte del resultado del diseño conceptual y da como resultado una **descripción de la estructura de la base de datos** en términos de las estructuras de datos que puede procesar un tipo de SGBD. El diseño lógico depende del tipo de SGBD que se vaya a utilizar, se adapta a la tecnología que se debe emplear, pero no depende del producto concreto. En el caso de bases de datos convencionales relacionales (basadas en SQL para entendernos), **el diseño lógico consiste en definir las tablas que existirán, las relaciones entre ellas,** normalizarlas, etc...

- El **diseño físico** parte del lógico y da como resultado una descripción de la implementación de una base de datos en memoria secundaria: las estructuras de almacenamiento y los métodos utilizados para tener un acceso eficiente a los datos. Aquí el objetivo es conseguir una mayor eficiencia, y se tienen en cuenta aspectos concretos del SGBD sobre el que se vaya a implementar. Por regla general esto es transparente para el usuario, aunque conocer cómo se implementa ayuda a optimizar el rendimiento y la escalabilidad del sistema.

El modelo relacional

En el modelo relacional las dos capas de diseño conceptual y lógico, se parecen mucho. Generalmente se implementan mediante **diagramas de Entidad/Relación** (modelo conceptual) y **tablas y relaciones** entre éstas (modelo lógico). Este es el modelo utilizado por los sistemas gestores de datos más habituales (SQL Server, Oracle, MySQL...).

> **Nota:** Aunque mucha gente no lo sabe, a las bases de datos relaciones se les denomina así porque almacenan los datos en forma de "Relaciones" o listas de datos, es decir, en lo que llamamos habitualmente "Tablas". Muchas personas se piensan que el nombre viene porque además las tablas se relacionan entre sí utilizando claves externas. No es así, y es un concepto que debemos tener claro. **(Tabla = Relación).**

El modelo relacional de bases de datos se rige por algunas normas sencillas:

- Todos los datos se representan en forma de **tablas** (también llamadas "relaciones", ver nota anterior). Incluso los resultados de consultar otras tablas. La tabla es además la unidad de almacenamiento principal.
- Las tablas están compuestas por filas (o registros) y columnas (o campos) que almacenan cada uno de los registros (la información sobre una entidad concreta, considerados una unidad).

- Las filas y las columnas, en principio, carecen de orden a la hora de ser almacenadas. Aunque en la implementación del diseño físico de cada SGBD esto no suele ser así. Por ejemplo, en SQL Server si añadimos una clave de tipo "Clustered" a una tabla haremos que los datos se ordenen físicamente por el campo correspondiente.
- El orden de las columnas lo determina cada consulta (que se realizan usando SQL).
- Cada tabla debe poseer una **clave primaria,** esto es, un **identificador único** de cada registro compuesto por una o más columnas.
- Para establecer una **relación entre dos tablas** es necesario incluir, en forma de columna, en una de ellas la clave primaria de la otra. A esta columna se le llama **clave externa.** Ambos conceptos de clave son extremadamente importantes en el diseño de bases de datos.

Basándose en estos principios se diseñan las diferentes bases de datos relacionales, definiendo un diseño conceptual y un diseño lógico, que luego se implementa en el diseño físico usando para ello el gestor de bases de datos de nuestra elección (por ejemplo SQL Server).

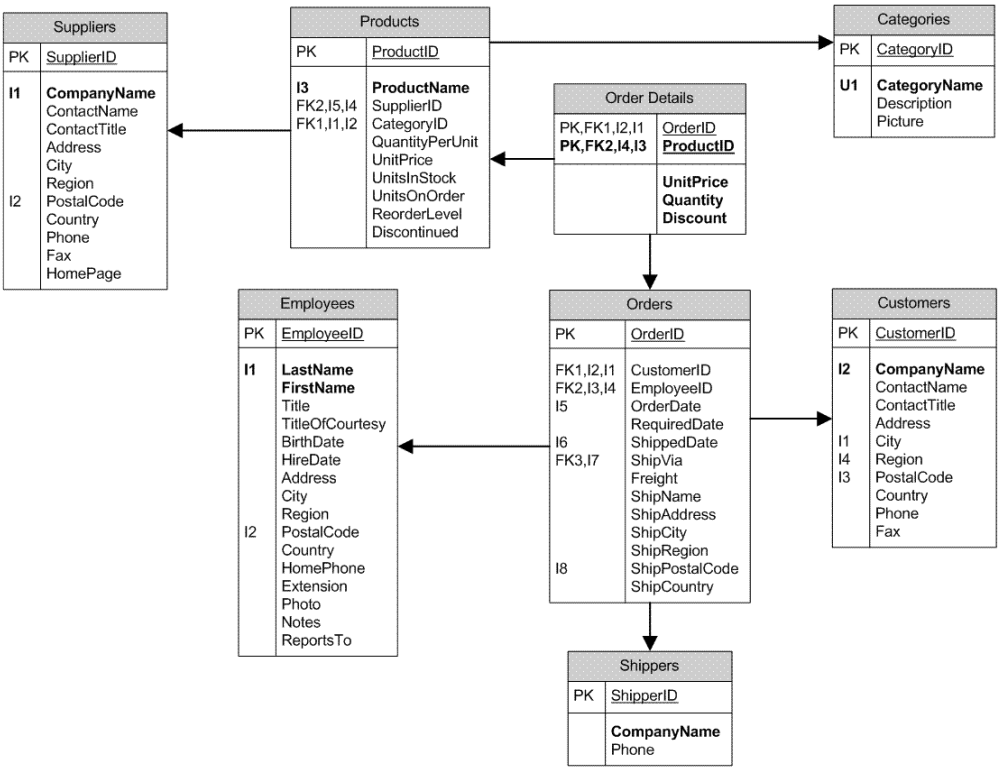
Por ejemplo, consideremos la conocida **base de datos Northwind** de Microsoft (ver siguiente capítulo).

Esta base de datos representa un **sistema sencillo de gestión de pedidos para una empresa ficticia.** Existen conceptos que hay que manejar como: proveedores, empleados, clientes, empresas de transporte, regiones geográficas, y por supuesto pedidos y productos.

El **diseño conceptual** de la base de datos para manejar toda esta información se puede ver en la figura de la página siguiente, denominada **diagrama Entidad/Relación** o simplemente **diagrama E-R.**

Como vemos en la figura (siguiente página), existen **tablas para representar cada una de estas entidades** del mundo real: Proveedores *(Suppliers)*, Productos, Categorías de productos, Empleados, Clientes, Transportistas *(Shippers),* y Pedidos *(Orders)* con sus correspondientes líneas de detalle *(Order Details)*.

Además **están relacionadas entre ellas** de modo que, por ejemplo, un producto pertenece a una determinada categoría (se relacionan por el campo *(CategoryID)* y un proveedor *(SupplierID),* y lo mismo con las demás tablas.



Cada tabla posee una serie de **campos que representan valores que queremos almacenar para cada entidad.** Por ejemplo, un producto posee los siguientes atributos que se traducen en los campos correspondientes para almacenar su información: Nombre *(ProductName)*, Proveedor *(SupplierID*, que identifica al proveedor), Categoría a la que pertenece *(CategoryID),* Cantidad de producto por cada unidad a la venta *(QuantityPerUnit)*, Precio unitario *(UnitPrice)*, Unidades que quedan en stock (UnitsInStock), Unidades de ese producto que están actualmente en pedidos *(UnitsOnOrder)*, qué cantidad debe haber para que se vuelva a solicitar más producto al proveedor *(ReorderLevel)* y si está descatalogado o no *(Discontinued)*.

Los campos marcados con **"PK"** indican aquellos que son **claves primarias**, es decir, que identifican de manera única a cada entidad. Por ejemplo, *ProductID* es el identificador único del producto, que será por regla general un número entero que se va incrementando cada vez que introducimos un nuevo producto (1, 2, 3, etc..).

Los campos marcados como **"FK"** son **claves foráneas o claves externas.** Indican campos que van a almacenar claves primarias de otras tablas de modo que se puedan

relacionar con la tabla actual. Por ejemplo, en la tabla de productos el campo *CategoryID* está marcado como "FK" porque en él se guardará el identificador único de la categoría asociada al producto actual. En otras palabras: ese campo almacenará el valor de la clave primaria (PK) de la tabla de categorías que identifica a la categoría en la que está ese producto.

Los campos marcados con indicadores que empiezan por **"I"** (ej: "I1") se refieren a **índices.** Los índices generan información adicional para facilitar la **localización más rápida de registros** basándose en esos campos. Por ejemplo, en la tabla de empleados **(Employees)** existe un índice "I1" del que forman parte los campos Nombre y Apellidos (en negrita además porque serán también valores únicos) y que indica que se va a facilitar la locación de los clientes mediante esos datos. También tiene otro índice "I2" en el campo del código postal para localizar más rápidamente a todos los clientes de una determinada zona.

Los campos marcados con indicadores que empiezan con **"U"** (por ejemplo U1) se refieren a campo que deben ser **únicos.** Por ejemplo, en la tabla de categorías el nombre de ésta *(CategoryName)* debe ser único, es decir, no puede haber -lógicamente- dos categorías con el mismo nombre.

> Como vemos, un diseño conceptual no es más que una representación formal y acotada de entidades que existen en el mundo real, así como de sus restricciones, y que están relacionadas con el dominio del problema que queremos resolver.

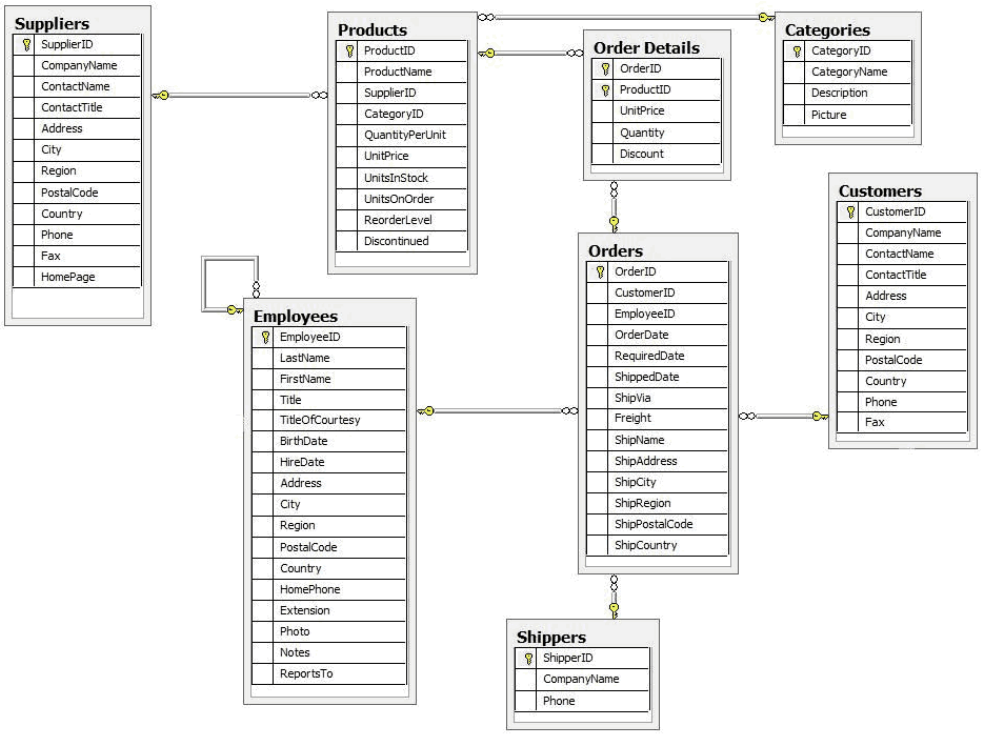
Modelo lógico

Una vez tenemos claro el modelo E-R debemos traducirlo a un modelo lógico directamente en el propio sistema gestor de bases de datos (Oracle, MySQL, SQL Server...). Si hemos utilizado alguna herramienta profesional para crear el diagrama E-R, seguramente podremos **generar automáticamente las instrucciones necesarias para**

crear la base de datos.

> La mayoría de los generadores de diagramas E-R (por ejemplo Microsoft Visio) ofrecen la capacidad de exportar el modelo directamente a los SGBD más populares.

Entonces, todo este modelo conceptual se traduce en un **modelo lógico** que trasladaremos a la base de datos concreta que estemos utilizando y que generalmente será muy parecido. Por ejemplo, este es el mismo modelo anterior, mostrado ya como tablas en un diagrama de SQL Server:



En este caso hemos creado cada tabla, una a una, siguiendo lo identificado en el diagrama E-R y estableciendo índices y demás elementos según las indicaciones de

cada uno de los campos. Además hemos decidido el mejor tipo de datos que podemos aplicar a cada campo (texto, números, fechas... que se almacenan para cada registro).

Su representación gráfica en la base de datos es muy similar, sin embargo el modelo físico (cómo se almacena esto físicamente), puede variar mucho de un SGBD a otro y según la configuración que le demos.

En resumen

Según Thomas H. Grayson, **un buen diseño de base de datos debe poseer siempre las siguientes cualidades,** aunque algunas puede llegar a ser contradictorias entre sí:

- 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.

Como hemos visto el diseño de una base de datos parte de un problema real que queremos resolver y se traduce en una serie de modelos, conceptual, lógico y físico, que debemos implementar.

El primero, **el diseño conceptual, es el que más tiempo nos va a llevar** pues debemos pensar muy bien cómo vamos a representar las entidades del mundo real que queremos representar, qué datos almacenaremos, cómo los relacionaremos entre sí, etc...

El diseño lógico es mucho más sencillo puesto que no es más que pasar el diseño anterior a una base de datos concreta. De hecho muchas herramientas profesionales nos ofrecen la generación automática del modelo, por lo que suele ser muy rápido.

El diseño físico por regla general recae en la propia base de datos, a partir del diseño lógico, aunque si dominamos bien esa parte elegiremos cuidadosamente **índices, restricciones o particiones** así como configuraciones para determinar cómo se almacenará físicamente esa información, en qué orden, cómo se repartirá físicamente en el almacenamiento, etc...

Acceso a datos

Cómo instalar paso a paso la base de datos Northwind

La **base de datos Northwind** es archi-conocida. Se lleva utilizando **más de una década para aprender a trabajar con bases de datos SQL Server** en infinidad de artículos, libros y cursos.

> Ya presentamos Northwind brevemente en el capítulo anterior, y vimos también su estructura de tablas y relaciones.

Su ventaja es que es una base de datos sencilla pero al mismo tiempo contiene gran cantidad de información, relaciones, etc... y simula las necesidades básicas de una empresa sencilla de importación de productos alimentarios. Contiene tablas para gestionar la información de productos, clientes, proveedores, personal, pedidos, transportistas, etc...

Microsoft la tiene todavía disponible para descarga. El problema de esta descarga es que es un archivo .msi que debemos instalar, y contiene una versión muy antigua destinada a trabajar con SQL Server 2000. En CodePlex podemos encontrar una versión más moderna de la base de datos que funciona sin problemas con las versiones más recientes de SQL Server.

Lo que ocurre es que, **incluso esta versión más reciente, tiene algunos problemas a la hora de crearla,** tanto con SQL Server Management Studio como con Visual Studio.

En el siguiente vídeo enseño muy paso a paso, y pensando en principiantes, cómo descargar e instalar la base de datos Nortwind, tanto con SQL Server Management Studio como con Visual Studio, resolviendo además los posibles problemas que nos podamos encontrar por el camino:

Cómo instalar Northwind con SQL Server Manageme...
Cómo instalar sin problemas
la base de datos Northwind
con SQL Server o con Visual Studio
www.campusmvp.es
campus MVP
Autor y tutor
José M. Alarcón
@jm_alarcon

Fundamentos de SQL

Cómo realizar consultas simples con SELECT

En los capítulos anteriores veíamos qué es el lenguaje SQL y sus diferentes subconjuntos de instrucciones. También, tratamos los fundamentos de diseño de una base de datos relacional. A continuación, aprenderemos los fundamentos de **consultas simples de datos con SELECT.**

Operaciones básicas de manipulación de datos en SQL

Como hemos visto, las instrucciones **DML** (*Data Manipulation Language – Lenguaje de Manipulación de Datos)* trabajan sobre los datos almacenados en nuestro SGBD, permitiendo consultarlos o modificarlos.

> En general a las operaciones básicas de manipulación de datos que podemos realizar con SQL se les denomina **operaciones CRUD** (de **C**reate, **R**ead, **U**pdate and **D**elete, o sea, Crear, Leer, Actualizar y Borrar, sería CLAB en español, pero no se usa). Lo verás utilizado de esta manera en muchos sitios, así que apréndete ese acrónimo.

Hay cuatro instrucciones para realizar estas tareas:

- **INSERT:** Inserta filas en una tabla. Se corresponde con la "C" de CRUD.
- **SELECT:** muestra información sobre los datos almacenados en la base

de datos. Dicha información puede pertenecer a una o varias tablas. Es la "R".

- **UPDATE:** Actualiza información de una tabla. Es, obviamente, la "U".
- **DELETE:** Borra filas de una tabla. Se correspontde con la "D".

Consulta de datos

Ahora nos vamos a centrar en la "R" de CRUD, es decir, en **cómo recuperar la información que nos interesa de dentro de una base de datos,** usando para ello el lenguaje de consulta o SQL. Ya nos preocuparemos luego de cómo llegamos a introducir los datos primeramente.

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). También es una de las instrucciones en la que con más frecuencia los motores de bases de datos incorporan cláusulas adicionales al estándar, que es el que veremos aquí.

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. Igual que cualquier tabla de la base de datos. Sin embargo esta tabla está en memoria mientras la utilicemos, y luego se descarta. Cada vez que ejecutamos la consulta se vuelve a calcular el resultado.

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 ]
```

A continuación analizaremos cada una de las partes de la consulta para entenderla mejor:

SELECT

Permite seleccionar las columnas que se van a mostrar y en el orden en que lo van a hacer. Simplemente es la instrucción que la base de datos interpreta como que vamos a solicitar información.

ALL / DISTINCT

ALL es el valor predeterminado, especifica que el conjunto de resultados puede incluir filas duplicadas. Por regla general nunca se utiliza.

DISTINCT especifica que el conjunto de resultados sólo puede incluir filas únicas. Es decir, si al realizar una consulta hay registros exactamente iguales que aparecen más de una vez, éstos se eliminan. Muy útil en muchas ocasiones.

Nombres de campos

Se debe especificar una lista de nombres de campos de la tabla que nos interesan y que por tanto queremos devolver. Normalmente habrá más de uno, en cuyo caso separamos cada nombre de los demás mediante comas.

Se puede anteponer el nombre de la tabla al nombre de las columnas, utilizando el formato Tabla.Columna. Además de nombres de columnas, en esta lista se pueden poner constantes, expresiones aritméticas, y funciones, para obtener campos calculados de manera dinámica.

Si queremos que nos devuelva todos los campos de la tabla utilizamos el comodín "*" (asterisco).

Los nombres indicados deben coincidir exactamente con los nombre de los campos de la tabla, pero si queremos que en nuestra tabla lógica de resultados tengan un nombre diferente podemos utilizar:

AS

Permite renombrar columnas si lo utilizamos en la cláusula SELECT, o renombrar tablas si lo utilizamos en la cláusula FROM. Es opcional. Con ello podremos crear diversos alias de columnas y tablas. Enseguida veremos un ejemplo.

FROM

Esta cláusula permite indicar las tablas o vistas de las cuales vamos a obtener la información. De momento veremos ejemplos para obtener información de una sola tabla.

Como se ha indicado anteriormente, también se pueden renombrar las tablas usando la instrucción "AS".

WHERE

Especifica la **condición de filtro** de las filas devueltas. Se utiliza cuando no se desea que se devuelvan todas las filas de una tabla, sino sólo las que cumplen ciertas condiciones. Lo habitual es utilizar esta cláusula en la mayoría de las consultas.

Condiciones

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 **diversos operadores** 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.

- **LIKE:** para la comparación de un modelo. 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 caracter). Con la combinación de estos caracteres podremos obtener múltiples patrones de búsqueda. Por ejemplo:
  - 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%'`

- **BETWEEN:** para un intervalo de valores. Por ejemplo:
  - Clientes entre el 30 y el 100: CodCliente BETWEEN 30 AND 100
  - Clientes nacidos entre 1970 y 1979: FechaNac BETWEEN '19700101' AND '19791231'

- **IN( ):** para especificar una relación de valores concretos. Por ejemplo: Ventas de los Clientes 10, 15, 30 y 75: `CodCliente IN(10, 15, 30, 75)`

Por supuesto es posible combinar varias condiciones simples de los operadores anteriores utilizando los operadores lógicos **OR, AND 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.

ORDER BY

Define el orden de las filas del conjunto de resultados. Se especifica el campo o campos (separados por comas) por los cuales queremos ordenar los resultados.

ASC / DESC

ASC es el valor predeterminado, especifica que la columna indicada en la cláusula ORDER BY se ordenará de forma ascendente, o sea, de menor a mayor. Si por el contrario se especifica DESC se ordenará de forma descendente (de mayor a menor).

Por ejemplo, para ordenar los resultados de forma ascendente por ciudad, y los que sean de la misma ciudad de forma descendente por nombre, utilizaríamos esta cláusula de ordenación:

```
… ORDER BY Ciudad, Nombre DESC …
```

Como a la columna *Ciudad* no le hemos puesto ASC o DESC se usará para la misma el valor predeterminado (que es ASC).

> OJO: Aunque al principio si aún no se está habituado, pueda dar la impresión de que se ordena por ambas columnas en orden descendente. Si es eso lo que queremos deberemos escribir ... ORDER BY Ciudad DESC, Nombre DESC ...

Algunos ejemplos

Para terminar este repaso a las consultas simples practicarlas un poco, veamos algunos ejemplos con la base de datos Northwind en SQL Server:

- Mostrar todos los datos de los Clientes de nuestra empresa:

```
SELECT * FROM Customers
```

- Mostrar apellido, ciudad y región (LastName, City, Region) de los empleados de USA (nótese el uso de AS para darle el nombre en español a los campos devueltos):

```
SELECT E.LastName AS Apellido, City AS Ciudad, Region
FROM Employees
WHERE Country = 'USA'
```

- Mostrar los clientes que no sabemos a qué región pertenecen (o sea, que no tienen asociada ninguna región) :

```
SELECT * FROM Customers WHERE Region IS NULL
```

- Mostrar las distintas regiones de las que tenemos algún cliente, accediendo sólo a la tabla de clientes:

```
SELECT DISTINCT Region FROM Customers WHERE Region IS
NOT NULL
```

- Mostrar los clientes que pertenecen a las regiones CA, MT o WA, ordenados por región ascendentemente y por nombre descendentemente.

```
CODE SELECT * FROM Customers
WHERE Region IN('CA', 'MT', 'WA')
ORDER BY Region, CompanyName DESC
```

- Mostrar los clientes cuyo nombre empieza por la letra "W":

```
SELECT * FROM Customers WHERE CompanyName LIKE 'W%'
```

- Mostrar los empleados cuyo código está entre el 2 y el 9:

```
SELECT * FROM Employees WHERE EmployeeID BETWEEN 2 AND 9'
```

- Mostrar los clientes cuya dirección contenga "ki":

```
SELECT * FROM Customers WHERE Address LIKE '%ki%'
```

- Mostrar las Ventas del producto 65 con cantidades entre 5 y 10, o que no tengan descuento:

```
SELECT * FROM [Order Details] WHERE (ProductID = 65 AND
Quantity BETWEEN 5 AND 10) OR Discount = 0
```

Nota: En SQL Server, para utilizar nombres de objetos con caracteres especiales se deben poner entre corchetes. Por ejemplo en la consulta anterior [Order Details] se escribe entre corchetes porque lleva un espacio en blanco en su nombre. En otros SGBDR se utilizan comillas dobles (Oracle, por ejemplo: "Order Details") y en otros se usan comillas simples (por ejemplo en MySQL).

Resumen

Con esto hemos visto los fundamentos de las consultas de lectura de datos con SQL. A continuación, vamos a complicar la cosa un poco y añadiremos sub-consultas y algunas instrucciones más complejas, como agrupaciones de datos y funciones de agregación.

Fundamentos de SQL

Consultas SELECT multi-tabla - JOIN

En el capítulo anterior sobre fundamentos de SQL vimos lo básico de crear consultas con la instrucción SELECT. A continuación vamos a complicar un poco la cosa **aprendiendo a realizar consultas en varias tablas** de la base de datos al mismo tiempo.

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 Northwind:

```
SELECT COUNT(*) FROM Customers
SELECT COUNT(*) FROM Orders
SELECT COUNT(*) FROM Customers, Orders
```

La primera instrucción devuelve 91 filas (los 91 clientes), la segunda 830 filas (los pedidos), y la tercera 75.530 (que son 830 x 91, es decir, la combinación de todas las filas de clientes y de pedidos).

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.

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 número de venta, el código y nombre del cliente y la fecha de venta en la base de datos Northwind:

```
SELECT OrderID, C.CustomerID, CompanyName, OrderDate
FROM Customers C, Orders O
WHERE C.CustomerID = O.CustomerID
```

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 del cliente, clave existente en ambas. Fíjate en como le hemos otorgado un alias a cada tabla (C y O 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:En el caso del ejemplo anterior quedaría de la siguiente forma:

```
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 un ejemplo en ambos formatos que involucra más tablas, en este caso las tablas de empleados, clientes y ventas:

```
SELECT OrderID, C.CustomerID, CompanyName, OrderDate
FROM Customers C, Orders O, Employees E
WHERE C.CustomerID = O.CustomerID AND O.EmployeeID =
E.EmployeeID
```

El segundo formato permite distinguir las condiciones que utilizamos para combinar las tablas y evitar el producto cartesiano, de las condiciones de filtro que tengamos que establecer.

Veamos un ejemplo como el anterior, pero ahora además necesitamos que el cliente sea de España o el vendedor sea el número 5.

En el primer formato tendríamos algo como esto:

```
SELECT OrderID, C.CustomerID, CompanyName, OrderDate
FROM Customers C, Orders O, Employees E
WHERE (C.CustomerID = O.CustomerID AND O.EmployeeID =
E.EmployeeID)
AND (C.Country = 'Spain' OR E.EmployeeID = 5)
```

Es decir, estamos mezclando en el WHERE las uniones de tablas, y las condiciones concretas de filtro de la consulta, quedando todo mucho más liado.

Sin embargo usando el segundo formato con JOIN, la consulta es mucho más clara:

```
SELECT OrderID, C.CustomerID, CompanyName, OrderDate
FROM Customers C
JOIN Orders O ON C.CustomerID = O.CustomerID JOIN Employees E
ON O.EmployeeID = E.EmployeeID
WHERE C.Country = 'Spain' OR E.EmployeeID = 5
```

Aquí se aprecia claramente que la utilización de JOIN simplifica la lectura y comprensión de la instrucción SQL, ya que no necesita el uso de paréntesis y tiene una condición WHERE más sencilla.

También podemos utilizar una misma tabla con dos alias diferentes para distinguirlas. Veamos un ejemplo, supongamos que tenemos una columna sueldo en la tabla de empleados, y queremos saber los empleados que tienen un sueldo superior al del empleado 5:

```
SELECT EmployeeID
FROM Employees E1 JOIN Employees E2 ON E1.Sueldo > E2.Sueldo
WHERE E2.EmployeeID = 5
```

Con esto hemos aprendido lo básico de trabajar con varias tablas y generar combinaciones de datos entre éstas.

A continuación, aprenderemos a ver los otros dos tipos de combinaciones que existen: **las combinaciones internas y las externas,** así como las **combinaciones de conjuntos de resultados** (uniones, intersecciones, etc...).

Fundamentos de SQL

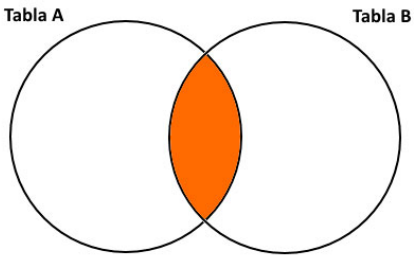
Consultas SELECT multi-tabla - Tipos de JOIN

Ya sabemos cómo funcionan las consultas multi-tabla basadas en JOIN. Ahora vamos a aprender 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 clientes y pedidos en Northwind, se devolverán los registros de todos los clientes que tengan al menos un pedido, relacionándolos por el ID de cliente.

> Esto **puede ocasionar la desaparición 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
```

Por ejemplo, para obtener en Northwind los clientes que tengan algún pedido, bastaría con escribir:

```
SELECT OrderID, C.CustomerID, CompanyName, OrderDate
FROM Customers C INNER JOIN Orders O ON C.CustomerID =
O.CustomerID
```

Que nos devolverá 830 registros. Hay dos pedidos en la tabla de Orders sin cliente asociado, como puedes comprobar, y éstos no se devuelven por no existir la relación entre ambas tablas.

> En realidad esto ya lo conocíamos puesto que en las combinaciones internas, el uso de la palabra **INNER** es opcional (por eso lo hemos puesto entre corchetes). Si simplemente indicamos la palabra JOIN y la combinación de columnas (como ya hemos visto en el capítulo anterior) el sistema sobreentiende que estamos haciendo una combinación interna. Lo hemos incluido por ampliar la explicación y por completitud.

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

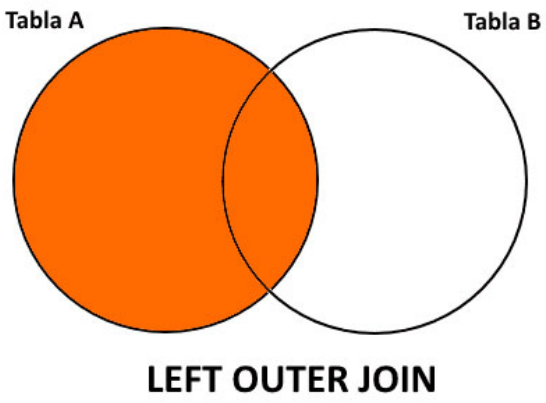
> En todas estas combinaciones externas **el uso de la palabra OUTER** es opcional. Si utilizamos LEFT, RIGHT o FULL y la combinación de columnas, el sistema sobreentiende que estamos haciendo una combinación externa.

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
```

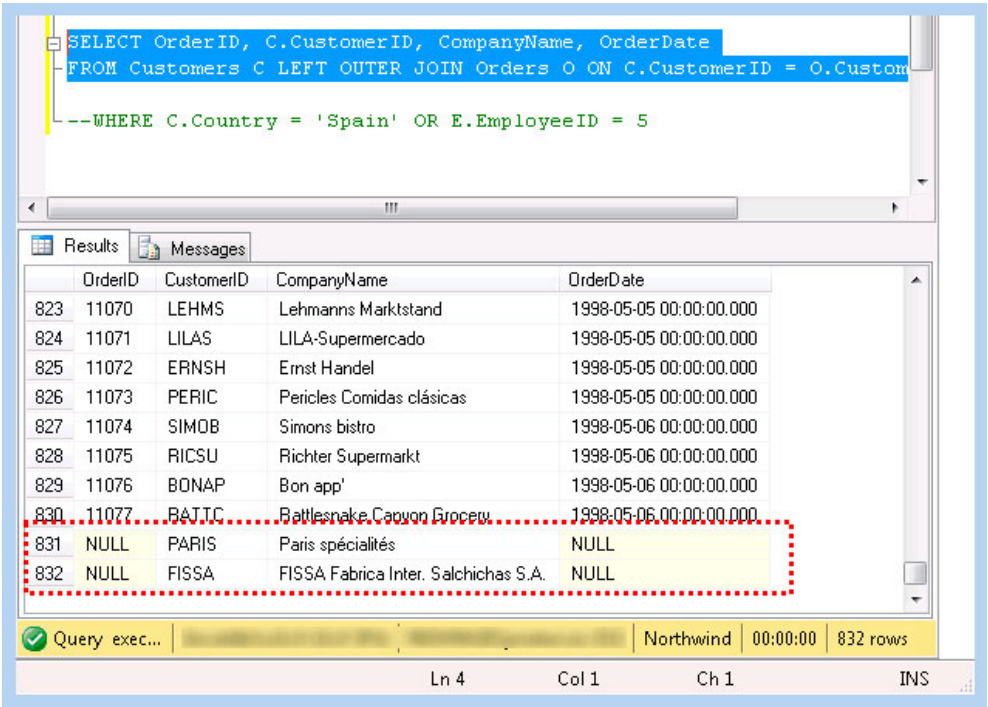
Esto se ilustra gráficamente de la siguiente manera:



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

```
SELECT OrderID, C.CustomerID, CompanyName, OrderDate
FROM Customers C LEFT JOIN Orders O ON C.CustomerID =
O.CustomerID
```

Obtendremos 832 registros ya que se incluyen **todos los clientes y sus pedidos, incluso aunque no tengan pedido alguno.** Los que no tienen pedidos carecen de la relación apropiada entre las dos tablas a partir del campo CustomerID. Sin embargo se añaden al resultado final dejando la parte correspondiente a los datos de la tabla de pedidos con valores nulos, como se puede ver en esta captura de SQL Server:

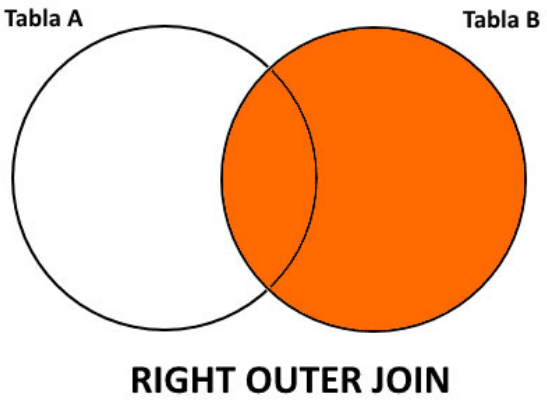


Variante RIGHT JOIN

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
```

El diagrama en este caso es complementario al anterior:



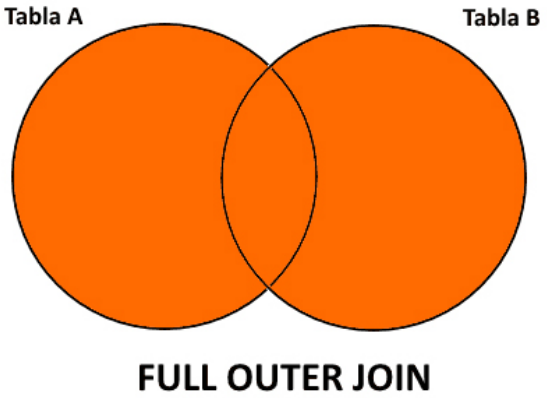
Si en nuestra base de datos de ejemplo queremos obtener t**odos los pedidos aunque no tengan cliente asociado**, junto a los datos de dichos clientes, escribiríamos:

```
SELECT OrderID, C.CustomerID, CompanyName, OrderDate
FROM Customers C RIGHT JOIN Orders O ON C.CustomerID =
O.CustomerID
```

En este caso se devuelven 830 registros que son todos los pedidos. Si hubiese algún pedido con el CustomerID vacío (nulo) se devolvería también en esta consulta (es decir, órdenes sin clientes), aunque en la base de datos de ejemplo no se da el caso.

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:



Es equivalente a obtener los registros comunes (con un INNER) y luego añadirle los de la tabla A que no tienen correspondencia en la tabla B, con los campos de la tabla vacíos, y los registros de la tabla B que no tienen correspondencia en la tabla A, con los campos de la tabla A vacíos.
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
```

Por ejemplo, en Northwind esta consulta:

```
SELECT OrderID, C.CustomerID, CompanyName, OrderDate
FROM Customers C FULL JOIN Orders O ON C.CustomerID =
O.CustomerID
```

Nos devuelve nuevamente 832 registros: los clientes y sus pedidos, los clientes sin pedido (hay 2) y los pedidos sin cliente (que en este caso son 0).

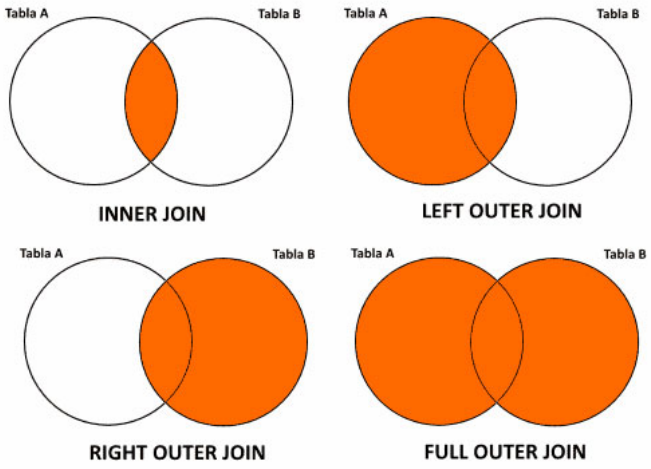
Fundamentos de SQL

Operaciones con conjuntos

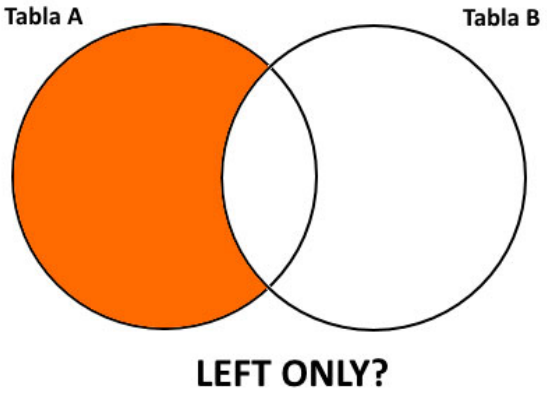
Hasta ahora hemos estudiado las consultas simples, las consultas multi-tabla y los diferentes tipos de JOIN en las consultas multi-tabla. Ahora vamos a aprender a operar con conjuntos, uniendo, intersecando y restando resultados.

Combinando consultas multi-tabla

Con las variantes INNER, LEFT, RIGHT y FULL de consultas multi-tabla somos capaces de obtener registros relacionados y los registros relacionados + los registros no relacionados en uno de los dos lados o en ambos, básicamente estas combinaciones de los datos de dos tablas:



Pero ¿qué pasa si queremos jugar con la parte común para conseguir combinaciones como éstas pero excluyendo los datos comunes?
Por ejemplo, en el caso de la base de datos Nortwind, si queremos obtener los clientes que NO tienen pedidos. Sería equivalente a esto en nuestros diagramas de Venn:



Es decir, sería equivalente a una hipotética cláusula LEFT ONLY (que no existe en SQL) en la que estamos excluyendo el resultado del INNER JOIN.
Dado que lo que queremos es encontrar a los que no tienen relación, es decir, aquellos cuyo campo de unión en el JOIN no existe en la tabla de la derecha, podemos usar una sintaxis como esta:

```
SELECT T1.Col1, T1.Col2, T1.Col3, T2.Col7
FROM Tabla1 T1 LEFT [OUTER] JOIN Tabla2 T2 ON T1.Col1 =
T2.Col1
WHERE T2.Col1 IS NULL
```

Es decir, basta con indicar que el campo en la tabla de la derecha es nulo, o sea, falla la relación por ese lado.

En nuestra base de datos de ejemplo si lanzamos esta consulta:

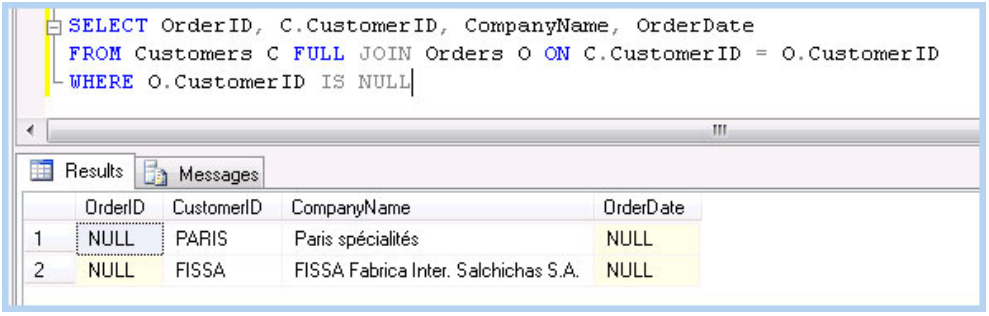
```
SELECT OrderID, C.CustomerID, CompanyName, OrderDate

FROM Customers C

FULL JOIN Orders O ON C.CustomerID = O.CustomerID

WHERE O.CustomerID IS NULL
```

Obtendremos todos los clientes que no tienen pedidos, que como sabemos de otras ocasiones son solamente dos:



Fíjate en como se obtienen los resultados con los campos correspondientes al pedido nulos.
Si solo nos interesara conocer qué clientes son estos sería fácil hacerlo con una consulta y su correspondiente sub-consulta, sin necesidad de usar un JOIN, de la siguiente manera:

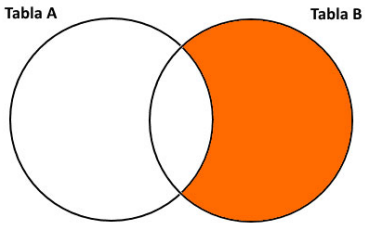
```
SELECT CustomerID, CompanyName FROM Customers

WHERE CustomerID NOT IN (SELECT DISTINCT CustomerID FROM
Orders
```

que nos devuelve la información que queremos pero la relación que buscamos es menos obvia y no involucra campos de la tabla de la derecha (a excepción de la clave externa, claro).
Exactamente del mismo modo pero cambiando la consulta por su "espejo" podríamos simular una hipotética función RIGHT ONLY de la siguiente manera:

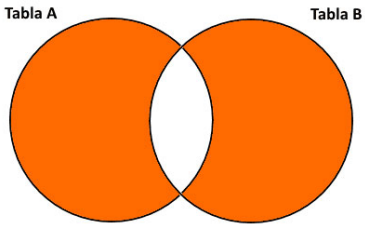
```
SELECT T1.Col1, T1.Col2, T1.Col3, T2.Col7
FROM Tabla1 T1
LEFT [OUTER] JOIN Tabla2 T2 ON T1.Col1 = T2.Col1
WHERE T1.Col1 IS NULL
```

que es equivalente al siguiente diagrama:



RIGHT ONLY?

Finalmente, y rizando el rizo, podríamos obtener únicamente todos los registros desparejados de la tabla de la izquierda y todos los desparejados de la tabla de la derecha para una hipotética operación EXCEPT INNER que no existe en SQL:



EXCEPT INNER?

simplemente combinando ambas condiciones vistas antes:

```
SELECT T1.Col1, T1.Col2, T1.Col3, T2.Col7
FROM Tabla1 T1
LEFT [OUTER] JOIN Tabla2 T2 ON T1.Col1 = T2.Col1
WHERE T1.Col1 IS NULL OR T2.Col1 IS NULL
```

Con esto tenemos contempladas todas las operaciones entre dos conjuntos de tablas relacionadas.

Operaciones de conjuntos entre tablas independientes

Además de lo visto hasta ahora es posible **combinar los resultados de dos consultas independientes** y fusionarlos en uno solo o realizar otras operaciones de conjuntos.

Por ejemplo podemos tomar el nombre y apellidos de todos los clientes de una tabla de clientes, y combinarlos con el nombre y los apellidos de todos los proveedores de una tabla de proveedores. No existe relación alguna entre ellos, pero son datos compatibles y podemos querer combinarlos.

Del mismo modo, y asumiendo que puede haber solapamiento entre ambas tablas, podríamos querer averiguar qué clientes tenemos que además son proveedores, o al contrario, qué proveedores no son clientes.

Veamos como...

La cláusula **UNION** de SQL permite unir las filas devueltas por dos instrucciones SELECT. Para ello **se debe cumplir que las columnas devueltas en ambas instrucciones coincidan en número y en tipo** de datos de cada una de ellas, ya que en caso contrario dará un error al ejecutarse.

Su sintaxis es:

```
SELECT Columnas FROM …
UNION [ALL]
SELECT Columnas FROM …
```

Si utilizamos **la opción ALL**, aparecerán todas las filas devueltas por ambas instrucciones SELECT, pero **si no la ponemos se eliminarán las filas repetidas.**

Veamos un ejemplo con la base de datos Northwind:

```
SELECT ShipCountry FROM Orders
UNION
SELECT Country FROM Customers
```

Esta consulta nos devolverá la lista de todos los países de destino de los pedidos, unidos a los países de ubicación de los clientes (que no tienen por qué coincidir). En este caso, si lanzamos la consulta, obtendremos 21 registros.

Sin embargo añadiéndole la opción ALL:

```
SELECT ShipCountry FROM Orders
UNION ALL
SELECT Country FROM Customers
```

nos devolverá todos los registros existentes, aunque estén repetidos, y obtendremos 921 filas como resultado (¡frente a 21 de antes!).

Si tuviésemos en vez de una sola tabla de ventas (*Orders*), una tabla de ventas por cada año (por ejemplo *Orders2001, Orders2002, Orders2003, Orders2004, Orders2005* y *Orders2006*). Si necesitamos un listado con el Nº de pedido, el nombre del empleado que la realizó, y la fecha, de todas las ventas del cliente cuyo código es 'ALFKI' a lo largo de todos esos años, podríamos combinar los resultados con UNION para obtener el listado consolidado:

```
SELECT O.OrderID, E.FirstName, O.OrderDate
FROM Orders2001 O
INNER JOIN Employees E ON O.EmployeeID = E.EmployeeID
   UNION ALL
SELECT O.OrderID, E.FirstName, O.OrderDate
FROM Orders2002 O
INNER JOIN Employees E ON O.EmployeeID = E.EmployeeID
   UNION ALL
SELECT O.OrderID, E.FirstName, O.OrderDate
FROM Orders2003 O
INNER JOIN Employees E ON O.EmployeeID = E.EmployeeID
   UNION ALL
SELECT O.OrderID, E.FirstName, O.OrderDate
FROM Orders2004 O
INNER JOIN Employees E ON O.EmployeeID = E.EmployeeID
   UNION ALL
SELECT O.OrderID, E.FirstName, O.OrderDate
FROM Orders2005 O
INNER JOIN Employees E ON O.EmployeeID = E.EmployeeID
   UNION ALL
SELECT O.OrderID, E.FirstName, O.OrderDate
FROM Orders2006 O
INNER JOIN Employees E ON O.EmployeeID = E.EmployeeID
```

> **Nota:** También podríamos hacer otras muchas cosas, como crear una vista basada en estas instrucciones SELECT y luego acceder a ella como si de una sola tabla se tratase, pero de momento no sabemos cómo hacer eso y de este modo vemos cómo usar UNION para conseguir el mismo resultado.

Además de esta instrucción, SQL incluye un par de instrucciones adicionales de gran utilidad para trabajar con conjuntos de tablas no relacionadas: **EXCEPT e INTERSECT.** Como cabría esperar por sus nombres, permiten respectivamente obtener **diferencias de conjuntos e intersecar conjuntos.**

> **Nota:** Aunque la función INTERSECT está ampliamente adoptada por la mayoría sistemas gestores de bases de datos relacionales, la instrucción EXCEPT está disponible en SQL Server, pero en el caso de Oracle o MySQL se llama MINUS. Aunque cambie el nombre la forma de usarla es idéntica.

Al igual que UNION estas dos operaciones se usan colocándolas entre dos consultas que deben ser compatibles.

- **INTERSECT** devuelve los valores distintos devueltos por las consultas y comunes a ambas, con lo que obtenemos una intersección (sólo los registros que están entre los resultados de ambas consultas).

- **EXCEPT (o MINUS)** devuelve los valores de la primera consulta que no se encuentran en la segunda. Así podemos averiguar qué registros están en una consulta pero no en la otra, calculando la diferencia entre dos conjuntos de registros. Algo realmente útil en ocasiones y difícil de conseguir con instrucciones más simples.

Con esto hemos dado un repaso bastante amplio a las posibilidades de trabajo con consultas de resultados en SQL estándar, que podremos aplicar a prácticamente cualquier gestor de bases de datos relacionales.

Fundamentos de SQL

Agrupaciones y funciones de agregación

En este capítulo vamos a estudiar cómo generar resultados de consultas agrupados y con algunas operaciones de agregación aplicadas.

Funciones de agregación

Las funciones de agregación en SQL nos permiten efectuar operaciones sobre un conjunto de resultados, pero devolviendo un único valor agregado para todos ellos. Es decir, nos permiten obtener medias, máximos, etc... sobre un conjunto de valores.

Las funciones de agregación básicas que soportan todos los gestores de datos son las siguientes:

- **COUNT:** devuelve el número total de filas seleccionadas por la consulta.
- **MIN:** devuelve el valor mínimo del campo que especifiquemos.
- **MAX:** devuelve el valor máximo del campo que especifiquemos.
- **SUM:** suma los valores del campo que especifiquemos. Sólo se puede utilizar en columnas numéricas.
- **AVG:** devuelve el valor promedio del campo que especifiquemos. Sólo se puede utilizar en columnas numéricas.

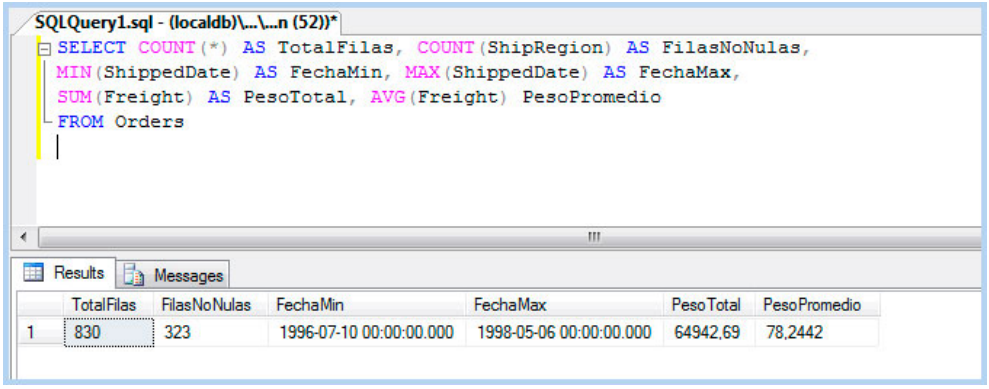
> Las funciones anteriores son las básicas en SQL, pero cada sistema gestor de bases de datos relacionales ofrece su propio conjunto, más amplio, con otras funciones de agregación particulares. Puedes consultar las que ofrecen SQL Server, Oracle y MySQL.

Todas estas funciones se aplican a una sola columna, que especificaremos entre paréntesis, excepto la función COUNT, que se puede aplicar a una columna o indicar un "*". La diferencia entre poner el nombre de una columna o un "*", es que en el primer caso no cuenta los valores nulos para dicha columna, y en el segundo si.

Así, por ejemplo, si queremos obtener algunos datos agregados de la tabla de pedidos de la base de datos de ejemplo Northwind, podemos escribir una consulta simple como la siguiente:

```
SELECT COUNT(*) AS TotalFilas, COUNT(ShipRegion) AS
FilasNoNulas, MIN(ShippedDate) AS FechaMin, MAX(ShippedDate)
AS FechaMax, SUM(Freight) AS PesoTotal,
AVG(Freight) PesoPromedio
FROM Orders
```

y obtendríamos el siguiente resultado en el entorno de pruebas:



De esta manera sabremos que existen en total 830 pedidos en la base de datos, 323 registros que tienen asignada una zona de entrega, la fecha del pedido más antiguo (el 10 de julio de 1996), la fecha del pedido más reciente (el 6 de mayo de 1998 ¡los datos de ejemplo son muy antiguos!), el total de peso enviado entre todos los pedidos (64.942,69 Kg o sea, más de 64 toneladas) y el peso promedio del los envíos (78,2442Kg). No está mal para una consulta tan simple.

Como podemos observar del resultado de la consulta anterior, **las funciones de agregación devuelven una sola fila**, salvo que vayan unidas a la cláusula GROUP BY, que veremos a continuación.

Agrupando resultados

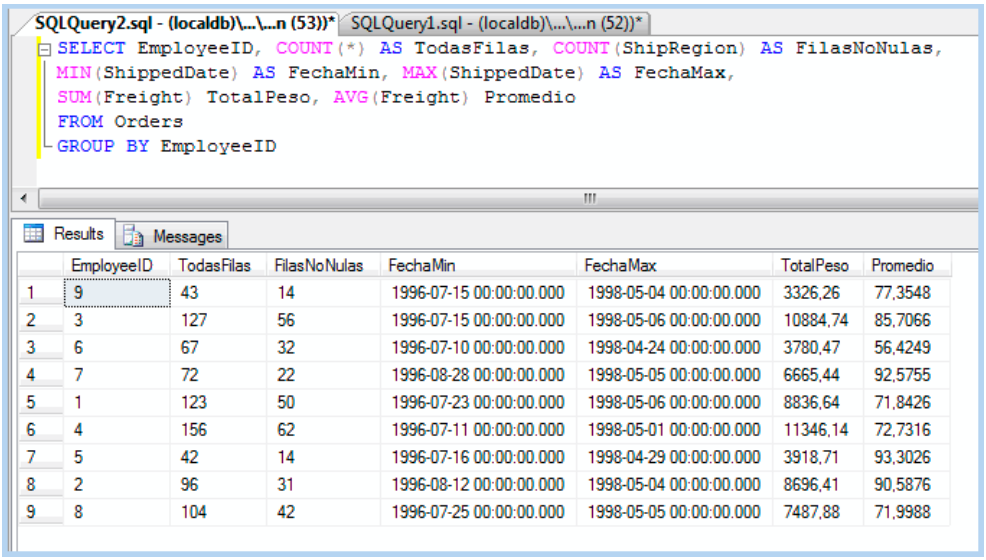
La cláusula GROUP BY unida a un SELECT **permite agrupar filas según las columnas que se indiquen como parámetros,** y se suele utilizar en conjunto con las funciones de agrupación, **para obtener datos resumidos y agrupados** por las columnas que se necesiten.

Hemos visto en el ejemplo anterior que obteníamos sólo una fila con los datos indicados **correspondientes a toda la tabla**. Ahora vamos a ver con otro ejemplo cómo obtener datos correspondientes a diversos grupos de filas, concretamente agrupados por cada empleado:

```
SELECT EmployeeID, COUNT(*) AS TotalPedidos, COUNT(ShipRegion)
AS FilasNoNulas, MIN(ShippedDate) AS FechaMin,
MAX(ShippedDate) AS FechaMax, SUM(Freight) PesoTotal,
AVG(Freight) PesoPromedio
FROM Orders
GROUP BY EmployeeID
```

En este caso obtenemos los mismos datos pero agrupándolos por empleado, de modo que para cada empleado de la base de datos sabemos cuántos pedidos ha

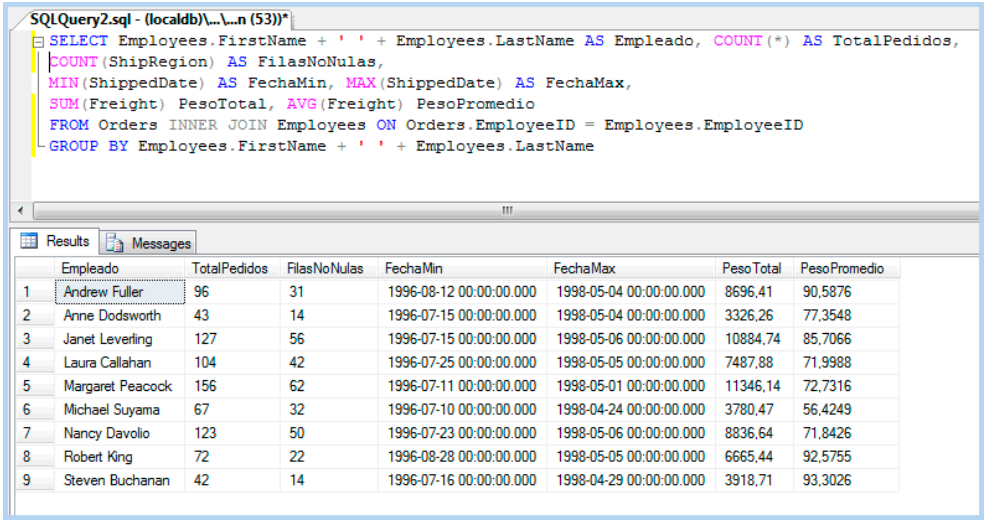
realizado, cuándo fue el primero y el último, etc...:



```
SELECT EmployeeID, COUNT(*) AS TodasFilas, COUNT(ShipRegion) AS FilasNoNulas,
MIN(ShippedDate) AS FechaMin, MAX(ShippedDate) AS FechaMax,
SUM(Freight) TotalPeso, AVG(Freight) Promedio
FROM Orders
GROUP BY EmployeeID
```

	EmployeeID	TodasFilas	FilasNoNulas	FechaMin	FechaMax	TotalPeso	Promedio
1	9	43	14	1996-07-15 00:00:00.000	1998-05-04 00:00:00.000	3326,26	77,3548
2	3	127	56	1996-07-15 00:00:00.000	1998-05-06 00:00:00.000	10884,74	85,7066
3	6	67	32	1996-07-10 00:00:00.000	1998-04-24 00:00:00.000	3780,47	56,4249
4	7	72	22	1996-08-28 00:00:00.000	1998-05-05 00:00:00.000	6665,44	92,5755
5	1	123	50	1996-07-23 00:00:00.000	1998-05-06 00:00:00.000	8836,64	71,8426
6	4	156	62	1996-07-11 00:00:00.000	1998-05-01 00:00:00.000	11346,14	72,7316
7	5	42	14	1996-07-16 00:00:00.000	1998-04-29 00:00:00.000	3918,71	93,3026
8	2	96	31	1996-08-12 00:00:00.000	1998-05-04 00:00:00.000	8696,41	90,5876
9	8	104	42	1996-07-25 00:00:00.000	1998-05-05 00:00:00.000	7487,88	71,9988

De hecho nos resultaría muy fácil cruzarla con la tabla de empleados, usando lo aprendido sobre consultas multi-tabla, y que se devolvieran los mismos resultados con el nombre y los apellidos de cada empleado:



```
SELECT Employees.FirstName + ' ' + Employees.LastName AS Empleado, COUNT(*) AS TotalPedidos,
COUNT(ShipRegion) AS FilasNoNulas,
MIN(ShippedDate) AS FechaMin, MAX(ShippedDate) AS FechaMax,
SUM(Freight) PesoTotal, AVG(Freight) PesoPromedio
FROM Orders INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
GROUP BY Employees.FirstName + ' ' + Employees.LastName
```

	Empleado	TotalPedidos	FilasNoNulas	FechaMin	FechaMax	PesoTotal	PesoPromedio
1	Andrew Fuller	96	31	1996-08-12 00:00:00.000	1998-05-04 00:00:00.000	8696,41	90,5876
2	Anne Dodsworth	43	14	1996-07-15 00:00:00.000	1998-05-04 00:00:00.000	3326,26	77,3548
3	Janet Leverling	127	56	1996-07-15 00:00:00.000	1998-05-06 00:00:00.000	10884,74	85,7066
4	Laura Callahan	104	42	1996-07-25 00:00:00.000	1998-05-05 00:00:00.000	7487,88	71,9988
5	Margaret Peacock	156	62	1996-07-11 00:00:00.000	1998-05-01 00:00:00.000	11346,14	72,7316
6	Michael Suyama	67	32	1996-07-10 00:00:00.000	1998-04-24 00:00:00.000	3780,47	56,4249
7	Nancy Davolio	123	50	1996-07-23 00:00:00.000	1998-05-06 00:00:00.000	8836,64	71,8426
8	Robert King	72	22	1996-08-28 00:00:00.000	1998-05-05 00:00:00.000	6665,44	92,5755
9	Steven Buchanan	42	14	1996-07-16 00:00:00.000	1998-04-29 00:00:00.000	3918,71	93,3026

En este caso fíjate en cómo hemos usado la expresión `Employees.FirstName + '' + Employees.LastName` como parámetro en GROUP BY para que nos agrupe por un campo compuesto (en SQL Server no podemos usar alias de campos para las agrupaciones). De esta forma tenemos casi un informe preparado con una simple consulta de agregación.

> **Importante:** Es muy importante tener en cuenta que cuando utilizamos la cláusula GROUP BY, los únicos campos que podemos incluir en el SELECT sin que estén dentro de una función de agregación, son los que vayan especificados en el GROUP BY..

La cláusula GROUP BY se puede utilizar con más de un campo al mismo tiempo. Si indicamos más de un campo como parámetro nos devolverá la información agrupada por los registros que tengan el mismo valor en los campos indicados.

Por ejemplo, si queremos conocer la cantidad de pedidos que cada empleado ha enviado a través de cada transportista, podemos escribir una consulta como la siguiente:

```
SELECT Employees.FirstName + ' ' + Employees.LastName AS
Empleado, Shippers.CompanyName AS Transportista,
COUNT(Orders.OrderID)AS NumPedidos
FROM Orders INNER JOIN Shippers ON Orders.ShipVia =
Shippers.ShipperID
INNER JOIN Employees ON Orders.EmployeeID=Employees.EmployeeID
GROUP BY Employees.FirstName + ' ' + Employees.LastName,
Shippers.CompanyName
```

Con el siguiente resultado:

SQLQuery4.sql - (localdb)\...\...n (54))*

```
SELECT Employees.FirstName + ' ' + Employees.LastName AS Empleado,
Shippers.CompanyName AS Transportista,
COUNT(Orders.OrderID)AS NumPedidos
FROM Orders INNER JOIN Shippers ON Orders.ShipVia = Shippers.ShipperID
INNER JOIN Employees ON Orders.EmployeeID=Employees.EmployeeID
GROUP BY Employees.FirstName + ' ' + Employees.LastName, Shippers.CompanyName
```

Results | Messages

	Empleado	Transportista	NumPedidos
1	Andrew Fuller	Federal Shipping	25
2	Anne Dodsworth	Federal Shipping	14
3	Janet Leverling	Federal Shipping	46
4	Laura Callahan	Federal Shipping	29
5	Margaret Peacock	Federal Shipping	40
6	Michael Suyama	Federal Shipping	19
7	Nancy Davolio	Federal Shipping	41
8	Robert King	Federal Shipping	28
9	Steven Buchanan	Federal Shipping	13
10	Andrew Fuller	Speedy Express	35
11	Anne Dodsworth	Speedy Express	10
12	Janet Leverling	Speedy Express	36
13	Laura Callahan	Speedy Express	27
14	Margaret Peacock	Speedy Express	46

Así, sabremos que Andrew Fuller envió 25 pedidos con Federal Shipping, y 35 con Federal Express.

El utilizar la cláusula **GROUP BY no garantiza que los datos se devuelvan ordenados.** Suele ser una práctica recomendable **incluir una cláusula ORDER BY** por las mismas columnas que utilicemos en GROUP BY, especificando el orden que no s interese. Por ejemplo, en el caso anterior.

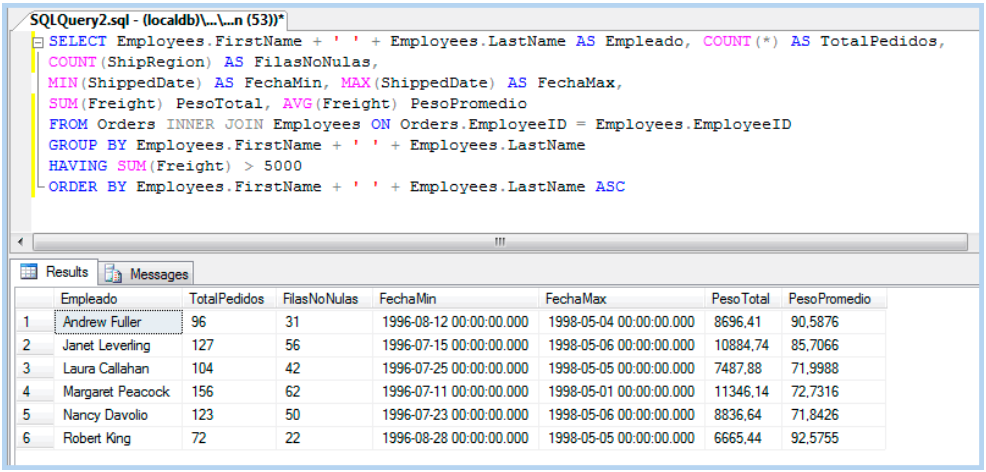
Existe una cláusula especial, parecida a la WHERE que ya conocemos que nos permite especificar las condiciones de filtro para los diferentes grupos de filas que devuelven estas consultas agregadas. Esta cláusula es **HAVING.**

> HAVING es muy similar a la cláusula WHERE, pero en vez de afectar a las filas de la tabla, afecta a los grupos obtenidos.

Por ejemplo, si queremos repetir la consulta de pedidos por empleado de hace un rato, pero obteniendo solamente aquellos que hayan enviado más de 5.000 Kg de producto, y ordenados por el nombre del empleado, la consulta sería muy sencilla usando HAVING y ORDER BY:

```
SELECT Employees.FirstName + ' ' + Employees.LastName AS
Empleado, COUNT(*) AS TotalPedidos, COUNT(ShipRegion) AS
FilasNoNulas, MIN(ShippedDate) AS FechaMin,
MAX(ShippedDate) AS FechaMax, SUM(Freight) PesoTotal,
AVG(Freight) PesoPromedio
FROM Orders INNER JOIN Employees ON Orders.EmployeeID =
Employees.EmployeeID
GROUP BY Employees.FirstName + ' ' + Employees.LastName
HAVING SUM(Freight) > 5000
ORDER BY Employees.FirstName + ' ' + Employees.LastName ASC
```

Ahora obtenemos los resultados agrupados por empleado también, pero solo aquellos que cumplan la condición indicada (o condiciones indicadas, pues se pueden combinar). Antes nos salían 9 empleados, y ahora solo 6 pues hay 3 cuyos envíos totales son muy pequeños:



	Empleado	TotalPedidos	FilasNoNulas	FechaMin	FechaMax	PesoTotal	PesoPromedio
1	Andrew Fuller	96	31	1996-08-12 00:00:00.000	1998-05-04 00:00:00.000	8696,41	90,5876
2	Janet Leverling	127	56	1996-07-15 00:00:00.000	1998-05-06 00:00:00.000	10884,74	85,7066
3	Laura Callahan	104	42	1996-07-25 00:00:00.000	1998-05-05 00:00:00.000	7487,88	71,9988
4	Margaret Peacock	156	62	1996-07-11 00:00:00.000	1998-05-01 00:00:00.000	11346,14	72,7316
5	Nancy Davolio	123	50	1996-07-23 00:00:00.000	1998-05-06 00:00:00.000	8836,64	71,8426
6	Robert King	72	22	1996-08-28 00:00:00.000	1998-05-05 00:00:00.000	6665,44	92,5755

Ya nos falta muy poco para dominar por completo las consultas de selección de datos en cualquier sistema gestor de bases de datos relacionales. El siguiente paso es aprender cómo realizar algunas consultas que implican el uso de sub-consultas o que aplican algunas sintaxis especiales para utilizar subconjuntos de datos.

Fundamentos de SQL

Funciones escalares en consultas de selección

Al igual que en cualquier otro lenguaje de programación, SQL dispone de una serie de funciones que nos facilitan la obtención de los resultados deseados. Éstas se pueden utilizar en las cláusulas SELECT, WHERE y ORDER BY que ya hemos estudiado. Otra característica a tener en cuenta es que se pueden anidar, es decir, una función puede llamar a otra función.

En el lenguaje SQL estándar existen básicamente **5 tipos de funciones: aritméticas**, de **cadenas** de caracteres, de **fechas,** de **conversión**, y **otras funciones** diversas que no se pueden incluir en ninguno de los grupos anteriores.

Es muy importante tener en cuenta que, habitualmente, cualquier sistema gestor de bases de datos relacionales (SGBDR) intenta incluir la mayoría de las funciones correspondientes al estándar ANSI SQL, y que además **suelen incluir un conjunto adicional de funciones propias.** Incluso de una versión a la siguiente dentro de un mismo producto, es común que aparezcan nuevas funciones.

En ese artículo vamos a explicar algunas funciones comunes. Dejamos que investigues y estudies las funciones propias del producto que te interese: SQL Server, Oracle, MySQL...

Aquí nos vamos a limitar a citar algunas de las más significativas, incluyendo una breve descripción de las mismas.

Funciones aritméticas

- **ABS(n):** Devuelve el valor absoluto de "n".
- **ROUND(m, n):** Redondea el número "m" con el número de decimales indicado en "n", si no se indica "n" asume cero decimales.
- **SQRT(n):** Devuelve la raíz cuadrada del parámetro que se le pase.
- **POWER(m, n):** Devuelve la potencia de "m" elevada el exponente "n".

Funciones de cadenas

- **LOWER(c):** Devuelve la cadena "c" con todas las letras convertidas a minúsculas.
- **UPPER(c):** Devuelve la cadena "c" con todas las letras convertidas a mayúsculas.
- **LTRIM(c):** Elimina los espacios por la izquierda de la cadena "c".
- **RTRIM(c):** Elimina los espacios por la derecha de la cadena "c".
- **REPLACE(c, b, s):** Sustituye en la cadena "c" el valor buscado "b" por el valor indicado en "s".
- **REPLICATE(c, n):** Devuelve el valor de la cadena "c" el número de veces "n" indicado.
- **LEFT(c, n):** Devuelve "n" caracteres por la izquierda de la cadena "c".
- **RIGHT(c, n):** Devuelve "n" caracteres por la derecha de la cadena "c".
- **SUBSTRING(c, m, n)**: Devuelve una sub-cadena obtenida de la cadena "c", a partir de la posición "m" y tomando "n" caracteres.
- **SOUNDEX(c):** Devuelve una cadena con la representación fonética para indexación en inglés (algoritmo Soundex) de la cadena "c".

Funciones de manejo de fechas

- **YEAR(d):** Devuelve el año correspondiente de la fecha "d".
- **MONTH(d):** Devuelve el mes de la fecha "d".
- **DAY(d):** Devuelve el día del mes de la fecha "d".
- **DATEADD(f, n, d):** Devuelve una fecha "n" periodos (días, meses años, según lo indicado) superior a la fecha "d". Si se le pasa un número "n" negativo, devuelve una fecha "n" periodos inferior. De gran utilidad en consultas.

Funciones de conversión

Esta funciones **suelen ser específicas de cada gestor de datos,** ya que cada SGBDR utiliza nombres diferentes para los distintos tipos de datos (aunque existen similitudes se dan muchas diferencias).

Las funciones de conversión nos permiten cambiar valores de un tipo de datos a otro. Por ejemplo si tenemos una cadena y sabemos que contiene una fecha, podemos convertirla al tipo de datos fecha. Así, por ejemplo:

- En **SQL Server** tenemos funciones como CAST, CONVERT o PARSE, que en función de lo que especifiquemos en sus parámetros convertirán los datos en el tipo que le indiquemos. Ver funciones de conversión de SQL Server.
- En **Oracle** tenemos funciones como TO_CHAR, TO_DATE, TO_NUMBER. Ver funciones de conversión de Oracle.
- En **MySQL** solucionamos la mayor parte de las conversiones con la función CAST. Ver conversiones en MySQL.

Algunos ejemplos simples

Una vez vistas estas funciones básicas, vamos a poner ejemplos sencillos de la sintaxis de alguna de ellas:

```
SELECT ABS(100), ABS(-100), ROUND(10.12345, 2), SQRT(9)
```

```
SELECT UPPER(LTRIM(RTRIM(' PEPE  '))), REPLICATE('AB', 20),
SUBSTRING('ABCDEFGHI', 3, 5)
```

```
SELECT MONTH(CAST('20060715' AS DATETIME)), DATEADD(dd, 10,
GETDATE())
```

> **IMPORTANTE:** En función del SGBDR que estemos utilizando podremos utilizar los ejemplos de diferente forma. Por ejemplo **SQL Server** y **MySQL** permiten utilizarlas y mostrar su resultado incluyendo una cláusula SELECT sin FROM ni ninguna otra cláusula, como en las líneas anteriores. **Oracle** sin embargo no permite esto, pero tiene una tabla especial llamada DUAL que sólo tiene una fila y una columna y está pensada para estos casos. En el caso de Oracle pues, podemos poner el código que aparece en los ejemplos anteriores y añadirle siempre " FROM Dual".

A continuación vamos a poner algunos ejemplos de uso de funciones basados ya en las tablas de la base de ejemplo Northwind para SQL Server:

1- Obtener el número de pedido, la fecha y la fecha de pago (suponiendo que esta última es 30 días después de la fecha de venta), para las ventas realizadas en los últimos 15 días:

```
SELECT OrderID, OrderDate, DATEADD(dd, 30, OrderDate)
FechaPago
FROM Orders
WHERE OrderDate BETWEEN DATEADD(dd, -15, GETDATE()) AND
GETDATE()
ORDER BY OrderDate DESC
```

2- Obtener el importe de cada una de las ventas de la base de datos, redondeado a dos decimales:

```
SELECT OrderID, ROUND(UnitPrice * Quantity, 2) as Importe
FROM [Order Details]
```

3- Mostrar para cada empresa cliente, su identificador/clave primaria, las tres primeras letras de su nombre, su nombre y el teléfono:

```
SELECT CustomerID, LEFT(CompanyName, 3), CompanyName,
Phone FROM Customers
```

Con ayuda de estas funciones, usadas tanto para devolver información como para filtrarla, podemos conseguir una gran flexibilidad en las consultas que realicemos.

Fundamentos de SQL

Fundamentos de SQL: Inserción de datos - INSERT

Ya hemos visto cómo extraer información de una base de datos. Sin embargo para de 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.

Veamos la sintaxis para cada uno de estos casos:

Insertando registros uno a uno

```
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 l**a 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 la práctica se suelen especificar siempre por claridad y para evitar errores.

Por otro lado, los valores se deben corresponder con cada uno de los campos que aparecen en la lista de campos, tanto en el tipo de dato que contienen como en el orden en el que se van a asignar. Es decir, si se indican una serie de campos en un orden determinado, la lista de valores debe especificar los valores a almacenar en dichos campos, **en el mismo orden exactamente.** Si un campo no está en la lista, se almacenará dentro de éste el valor NULL.

> Si un campo está definido como NOT NULL (es decir, que no admite nulos o valores vacíos), debemos especificarlo siempre en la lista de campos a insertar. De no hacerlo así se producirá un error al ejecución la correspondiente instrucción INSERT.

Veamos unos ejemplos con la base de datos de ejemplo Northwind:

```
INSERT INTO Region VALUES(35, 'Madrid')
```

```
INSERT INTO Region (RegionID, RegionDescription)
VALUES(35, 'Madrid')
```

```
INSERT INTO Shippers (ShipperID, CompanyName)
VALUES(4, 'Mi Empresa')
```

En el tercer ejemplo no hemos asignado valores a la columna Phone, por tanto tomará automáticamente el valor NULL, salvo que tenga un valor **DEFAULT** asignado.

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, como *RegionID* o *ShipperID)* 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. Por ello, por ejemplo, para insertar una nueva región en la tabla de regiones de Northwind lo único que hay que hacer es indicar su nombre, así:

```
INSERT INTO Region (RegionDescription) VALUES('Madrid')
```

Ahorrándonos tener que indicar el identificador.

> **Nota:** Aunque las tablas estén relacionadas entre sí de manera unívoca (por ejemplo, cada cabecera de factura con sus líneas de factura), no es posible insertar de un golpe los registros de varias tablas. Es necesario siempre introducir los registros uno a uno y tabla a tabla.

Inserción masiva de filas partiendo de consultas

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 o, incluso en algunos SGBDR, de otra base de datos externa. 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.

Veámoslo mejor con unos ejemplos sencillos de Northwind. Supongamos que tenemos una tabla nueva llamada NewCustomers en la que hemos introducido previamente los datos de nuevos clientes que no están en el sistema (por ejemplo por que los hemos cargado desde un programa externo) y queremos agregar a nuestra tabla de clientes pre-existente los nuevos clientes que son del país "España":

```
INSERT INTO Customers
    SELECT * FROM NewCustomers WHERE Country = 'Spain'
```

Lo que se suele hacer en estos casos es crear primeramente la consulta de selección, que generalmente será más compleja y puede involucrar varias tablas que no tienen porque tener los mismos campos. Esta consulta la única condición que debe cumplir es que los campos devueltos tienen que ser del mismo tipo y orden que los que hay en la tabla donde insertamos o que los que indiquemos opcionalmente en la consulta de inserción.

Fundamentos de SQL

Actualización de datos - UPDATE

En el capítulo anterior hemos visto cómo insertar información dentro de cualquier base de datos usando instrucciones SQL estándar. Pero como nada está escrito en piedra, una vez que hayamos introducido la información, casi seguro que t**arde 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.

Veamos unos ejemplos con la base de datos de ejemplo Northwind en SQL Server (serían idénticos en otros sistemas gestores de bases de datos relacionales como Oracle o MySQL):

1- Corrige el apellido del empleado con identificador 5. El correcto es "Smith":

```
UPDATE Employees
SET LastName = 'Smith'
WHERE EmployeeID = 5
```

2- Elimina el valor del campo observaciones (Notes) de todos los empleados:

```
UPDATE Employees
SET Notes = NULL
```

Nótese como en este caso al carecer de una cláusula WHERE que restrinja la actualización **se actualizan todos los registros.**

Es muy importante incluir una cláusula WHERE en las instrucciones UPDATE salvo en casos muy concretos como el del ejemplo anterior. De no hacerlo se actualizarán todos los registros de la tabla como acabamos de ver. Es habitual que, por error, se omita esta cláusula y **se pierdan datos de forma irreversible** (salvo que dispongamos de copias de seguridad o tengamos la posibilidad de hacer un ROLLBACK (cancelación de las instrucciones) si no hemos finalizado la transacción).

3- Incrementa un 21% el precio de los productos pertenecientes la categoría que tiene el identificador 2:

```
UPDATE Products
SET UnitPrice = UnitPrice + (UnitPrice * 21 / 100)
WHERE CategoryID = 2
```

Subconsultas como apoyo a la actualización

Podemos incluir una o varias subconsultas dentro una sentencia UPDATE. Éstas pueden estar contenidas en la cláusula WHERE o formar parte también de la cláusula SET. En este último caso se deben seleccionar el mismo número de campos y con los tipos de datos apropiados, para que cada uno de los valores pueda ser almacenado en la columna que hay a la izquierda del signo igual "=", entre paréntesis, al lado de la instrucción SET.

Veamos unos ejemplos:

1- Para todos los productos de la categoría bebidas ('Beverages'), de la cual no conocemos su identificador, duplica su precio:

```
UPDATE Products
SET UnitPrice = UnitPrice * 2
WHERE CategoryID = (SELECT CategoryID FROM Categories
                    WHERE CategoryName = 'Beverages')
```

Fíjate en como hemos utilizado una subconsulta en el filtro de modo que se averigua el identificador de la categoría que nos interesa, a partir de su nombre.

2- Asignar a todos los productos de la categoría bebidas ('beverages'), el mismo precio que tiene el producto con número de identificador (clave primaria) igual a 5:

```
UPDATE Products
SET UnitPrice = (SELECT UnitPrice FROM Products WHERE
ProductID = 5)
WHERE CategoryID = (SELECT CategoryID FROM Categories
                    WHERE CategoryName = 'Beverages')
```

En resumen

Con el uso de esta sencilla instrucción UPDATE podemos actualizar cualquier dato o, lo que es más interesante, conjunto de datos que tengamos albergado en cualquier tabla de nuestra base de datos.

La única precaución que debemos tener es la de incluir una clausula WHERE apropiada que verdaderamente cualifique a los datos que queremos actualizar. El principal error en el uso de esta instrucción es precisamente ese, ya que podríamos acabar actualizando los registros equivocados, haciendo mucho daño en la base de datos. Para evitar problemas conviene realizar una consulta de selección precia que nos

permita comprobar que las condiciones utilizadas en el WHERE son las apropiadas. También es conveniente tratar de lanzar la actualización dentro de una transacción que nos permita volver atrás todo el proceso en caso de producirse una equivocación y afectar al número equivocado de registros.

Fundamentos de SQL

Eliminación de datos - DELETE

El siguiente paso es aprender 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 de un plumazo **todas las filas de la tabla.** Es habitual que, por error, se omita esta cláusula y se pierdan datos de forma irreversible (salvo, claro está, que dispongamos de copias de seguridad o tengamos **la** posibilidad de retroceder una transacción abierta).

Como siempre, hagamos unos ejemplos con la base de datos de pruebas Northwind:

1- Eliminar de la base de datos al empleado cuyo identificador interno es el 9:

```
DELETE Employees WHERE EmployeeID = 9
```

2- Borrar a los clientes cuyo apellido (LastName) contenga la palabra "Desconocido":

```
DELETE Customers WHERE LastName LIKE '%Desconocido%'
```

3- Eliminar todos los productos de la categoría bebidas ("beverages"):

```
DELETE Products WHERE CategoryID =
(SELECT CategoryID FROM Categories
     WHERE CategoryName = 'Beverages')
```

4- Eliminar todos los productos de la categoría bebidas ("beverages") cuyo precio unitario (UnitPrice) sea superior a 50:

```
DELETE Products WHERE CategoryID =
(SELECT CategoryID FROM Categories WHERE CategoryName =
'Beverages')
AND UnitPrice > 50
```

Como vemos, en estos dos últimos ejemplos hemos utilizado subconsultas dentro de la condición que determina los elementos a borrar, al igual que hicimos ya con la instrucción UPDATE.

Con esta instrucción, la más sencilla de utilizar, hemos visto las tres operaciones básicas sobre la base de datos que provocan cambios en los datos: inserción, actualización y borrado.

Fundamentos de SQL

Transacciones

Antes de nada una definición:

> Una **transacción** es una **unidad de trabajo** compuesta por diversas tareas, cuyo resultado final debe ser **que se ejecuten todas o ninguna** de ellas.

Por regla general en un sistema de base de datos todas las operaciones relacionadas entre sí que se ejecuten dentro un mismo flujo lógico de trabajo, deben ejecutarse en bloque. De esta manera si todas funcionan la operación conjunta de bloque tiene éxito, pero si falla cualquiera de ellas, deberán retrocederse todas las anteriores que ya se hayan realizado. De esta forma **evitamos que el sistema de datos quede en un estado incongruente.**

Por ejemplo, si vamos al banco y ordenamos una transferencia para pagar una compra que hemos realizado por Internet, el proceso en sí está formado por una conjuto (o bloque) de operaciones que deben ser realizadas para que la operación global tenga éxito:

1- Comprobar que nuestra cuenta existe es válida y está operativa.

2- Comprobar si hay saldo en nuestra cuenta.

3- Comprobar los datos de la cuenta del vendedor (que existe, que tiene posibilidad de recibir dinero, etc...).

4- Retirar el dinero de nuestra cuenta

5- Ingresar el dinero en la cuenta del vendedor

Dentro de este proceso hay cinco operaciones, las cuales deben tener éxito o fallar conjuntamente.

En el caso de las operaciones 4 y 5 que modifican datos es algo obvio que no puede funcionar una y fallar la otra. Si se retira el dinero de nuestra cuenta en el paso 4 y hay algún problema que evita que pueda continuar el proceso, el dinero habrá salido de nuestra cuenta pero no se ha anotado en la cuenta de destino porque se ha producido un error. De repente hay un dinero que ha desaparecido y la base de datos se encuentra en un estado inconsistente. Es evidente que esto no puede ocurrir.

Precisamente para evitar este tipo de situaciones existen las transacciones: **marcan bloques completos de operaciones y comprueban que, o se realizan todas, o que si hay algún problema se deshacen todas.**

En nuestro ejemplo de transferencia fallida, al producirse un error en el paso 5 se habría deshecho automáticamente la operación de retirada de dinero del paso 4 y toda la información habría quedado como antes de comenzar el proceso.

Otro tema importante relacionado con las transacciones es la **gestión de la concurrencia y los bloqueos.** En el ejemplo del banco los 3 primeros pasos realmente no implican modificación alguna de datos, por lo que si fallan no dejan la base de datos en un estado inconsistente ¿o quizá sí?. ¿Qué pasaría si al mismo tiempo que se está realizando nuestra transferencia, entra en nuestra cuenta un cargo diferido que teníamos pendiente? Sería posible que de repente nos quedásemos sin saldo para realizar la operación actual o, peor aún, que se anotasen mal ambos cargos en cuenta de modo que se "pisasen" dejando un saldo inconsistente. O puede que entre los pasos 3 y 5 la cuenta del vendedor de repente se haya cancelado, justo en medio de la operación. El dinero iría a parar a una cuenta no válida.

Para controlar el comportamiento de las transacciones en estos casos se definen diferentes **niveles de aislamiento de una transacción.**

Otro ejemplo más común: la creación de un pedido. En este proceso se debe

consultar antes la tabla de productos y ver si hay stock, se inserta un registro en la tabla de pedidos (quién hace el pedido, sus datos, fecha, importe total..), uno o varios registros en la tabla de detalles del pedido (qué productos se incluyen y su cantidad), y se actualiza la tabla de stock de productos. Si además se genera al mismo tiempo la factura, el proceso continua involucrando a varias tablas más (cabeceras de factura, líneas de factura). En este caso también es muy importante que se lleve la operación a cabo por completo o que se deshaga por completo, ya que sino nos encontraríamos con datos incoherentes.

Existen infinidad de posibilidades de que algo salga mal en cualquier proceso que involucre varias operaciones, y las posibilidades de fallo se multiplican a medida que aumenta la simultaneidad de acceso a la misma información. Por eso los sistemas de datos grandes son muy complejos.

Para todas estas situaciones nos ayudan las transacciones.

Propiedades ACID

Una transacción, para cumplir con su propósito y protegernos de todos los problemas que hemos visto, debe presentar las siguientes características:

- **Atomicidad:** las operaciones que componen una transacción deben considerarse como una sola.
- **Consistencia:** una operación nunca deberá dejar datos inconsistentes.
- **Aislamiento:** los datos "sucios" deben estar aislados, y evitar que los usuarios utilicen información que aún no está confirmada o validada. (por ejemplo: ¿sigue siendo válido el saldo mientras realizo la operación?)
- **Durabilidad:** una vez completada la transacción los datos actualizados ya serán permanentes y confirmados.

A estas propiedades se las suele conocer como propiedades ACID (de sus siglas en inglés: *Atomicity, Consistency, Isolation y Durability).*

Cómo definir transacciones

Por regla general en los gestores de datos relacionales modernos disponemos de tres tipos de transacciones según la forma de iniciarlas:

- **De confirmación automática**: el gestor de datos inicia una transacción automáticamente por cada operación que actualice datos. De este modo mantiene siempre la consistencia de la base de datos, aunque puede generar bloqueos.
- **Implícitas:** cuando el gestor de datos comienza una transacción automáticamente cada vez que se produce una actualización de datos, pero el que dicha transacción se confirme o se deshaga, lo debe indicar el programador.
- **Explícitas:** son las que iniciamos nosotros "a mano" mediante instrucciones SQ. somos nosotros, los programadores, los que indicamos qué operacio0nes va a abarcar.

Una **transacción explícita** se define de manera general con una instrucción que marca su inicio, y dos posibles instrucciones que marcan su final en función de si debe tener éxito o debe fracasar en bloque.

Cada sistema gestor de bases de datos tiene sus pequeñas particularidades, pero podemos escribir con un pseudo-código, la sintaxis de una transacción genérica:

```
BEGIN TRAN
      Operación 1...
      Si fallo: ROLLBACK TRAN

       Operación 2....
       Si fallo: ROLLBACK TRAN
...

      Operación N....
      Si fallo: ROLLBACK TRAN
COMMIT TRAN
```

Es decir, se define el comienzo de una transacción, se comprueban posibles errores en cada paso, echando atrás todo el proceso (se le suele llamar "hacer un Rollback"), o confirmando el conjunto de operaciones completas al final del todo si no ha habido problemas (en la jerga habitual se suele hablar de "hacer un Commit").

De hecho no suele ser necesario comprobar los errores por el camino ya que por regla general el gestor de datos si detecta un error en cualquiera de los pasos dentro de una transacción, realizará un *rollback* automático de toda la operación.

Transacciones en SQL Server

En SQL Server las instrucciones equivalentes a las genéricas que acabamos de ver son:

- **BEGIN TRANSACTION** o **BEGIN TRAN:** marca el inicio de una transacción. TRAN es un sinónimo de TRANSACTION y se suele usar más a menudo por abreviar.
- **ROLLBACK TRANSATION** o **ROLLBACK TRAN:** fuerza que se deshaga la transacción en caso de haber un problema o querer abandonarla. Cierra la transacción.
- **COMMIT TRANSACTION** O **COMMIT TRAN:** confirma el conjunto de operaciones convirtiendo los datos en definitivos. Marca el éxito de la operación de bloque y cierra la transacción.

Los **niveles de aislamiento** que nos ofrece SQL Server son:

- **SERIALIZABLE:** No se permitirá a otras transacciones la inserción, actualización o borrado de datos utilizados por nuestra transacción. Los bloquea mientras dura la misma.
- **REPEATABLE READ:** Garantiza que los datos leídos no podrán ser cambiados por otras transacciones, durante esa transacción.

- **READ COMMITED:** Una transacción no podrá ver los cambios de otras conexiones hasta que no hayan sido confirmados o descartados.
- **READ UNCOMMITTED:** No afectan los bloqueos producidos por otras conexiones a la lectura de datos.
- **SNAPSHOT:** Los datos seleccionados en la transacción se verán tal y como estaban al comienzo de la transacción, y no se tendrán en cuenta las actualizaciones que hayan sufrido por la ejecución de otras transacciones simultáneas.

La instrucción SET TRANSACTION ISOLATION LEVEL nos permite cambiar el nivel de aislamiento.

Puedes leer la documentación oficial sobre transacciones de SQL Server.

Transacciones en MySQL

Hay que tener en cuenta que MySQL es un gestor de datos relacional que soporta diversos motores de almacenamiento por debajo (al menos 20 la última vez que los contamos). De todos esos solamente unos pocos soportan transacciones. Los dos más comunes son el tradicional MyISAM y el más moderno InnoDB. De estos dos **solamente InnoDB soporta el uso de transacciones** (con razón myISAM es tan rápido: da mucha velocidad a costa de no ofrecer consistencia en las operaciones, lo cual puede ser muy útil para ciertos tipos de aplicaciones).

En MySQL InnoDB las instrucciones equivalentes a las genéricas son las siguientes:

- **START TRANSACTION** o **BEGIN**: marca el inicio de una transacción. Se suele usar más a menudo BEGIN porque es más corto.
- **ROLLBACK:** fuerza que se deshaga la transacción en caso de haber un problema o querer abandonarla. Cierra la transacción.
- **COMMIT:** confirma el conjunto de operaciones convirtiendo los datos en definitivos. Marca el éxito de la operación de bloque y cierra la transacción.

En cuanto a los **niveles de aislamiento** que nos ofrece MySQL son los siguientes:

- **SERIALIZABLE**
- **REPEATABLE READ**
- **READ COMMITED**
- **READ UNCOMMITTED**

La instrucción que nos permite cambiar el nivel de aislamiento es también **SET TRANSACTION ISOLATION LEVEL .**

Puedes leer la documentación oficial sobre transacciones de MySQL. Hay unas cuantas particularidades así que conviene leerlo detenidamente.

Transacciones en Oracle

En Oracle las instrucciones equivalentes a las genéricas son las siguientes:

- **START TRANSACTION** o **BEGIN:** marca el inicio de una transacción. Se suele usar más a menudo BEGIN porque es más corto.
- **ROLLBACK:** fuerza que se deshaga la transacción en caso de haber un problema o querer abandonarla. Cierra la transacción.
- **COMMIT:** confirma el conjunto de operaciones convirtiendo los datos en definitivos. Marca el éxito de la operación de bloque y cierra la transacción.

En cuanto a los **niveles de aislamiento** que nos ofrece Oracle son idénticos a los anteriores:

- **SERIALIZABLE**
- **REPEATABLE READ**
- **READ COMMITED**

READ UNCOMMITTED

La instrucción que nos permite cambiar el nivel de aislamiento es también **SET TRANSACTION ISOLATION LEVEL.** Es interesante leer al respecto este artículo de su documentación oficial sobre concurrencia de datos y consistencia.

Puedes leer la documentación oficial sobre transacciones de Oracle.

Esperamos que hayas disfrutado de los contenidos de este libro y que te hayan resultado útiles, recuerda que ofrecemos unos estupendos cursos online tutelados sobre acceso a datos. Concretamente, este curso de fundamentos de SQL o este de preparación del examen oficial 70-761 sería la continuación ideal a lo aprendido en este e-book.

¿Quieres más?

Aprende SQL Server desde cero con nosotros

Aprende paso a paso a tu ritmo, online, con ejemplos reales y **con un tutor experto** siempre a mano para resolver tus dudas.

Conviértete en un verdadero héroe de SQL Server.

Curso online

Consultas a bases de datos - Fundamentos del lenguaje SQL con SQL Server

Ver curso de SQL

¿Crees que se te puede quedar corto? Entonces échale un vistazo al curso de preparación del examen oficial 70-761 de **SQL Server y Lenguaje T-SQL**. Parte de este mismo curso para añadir contenidos más avanzados y de paso prepararte la certificación.

campus MVP

Ver todo el catálogo de cursos

¿Por qué aprender con nosotros? Porque **creamos cursos online de calidad contrastada** cuyos autores y tutores son reconocidos expertos del sector.

¿Quieres más razones? Haz click aquí